Универзитет у Београду
Математички факултет

Мастер рад

Стабло опсега и примене

Душан Слијепчевић, 1074/2011
d.slijepcevic@gmail.com

Ментор
Проф. др Миодраг Живковић

Септембар, 2017.

Садржај

1 Увод

Ортогонална претрага опсега је један од основних проблема у рачунарској геометрији са применама у базама података, рачунарској графици, мобилном рачунарству, географским информационим системима, CAD системима, итд. Разматра се следећи проблем: за дати фиксни скуп тачака у d-димензионом простору, омогућити ефикасно извршавање упита који проналазе тачке које леже унутар задатог ортогоналног d-димензионог опсега, односно d-квадра. Стабло опсега је једна од структура података која ефикасно решава овај проблем.

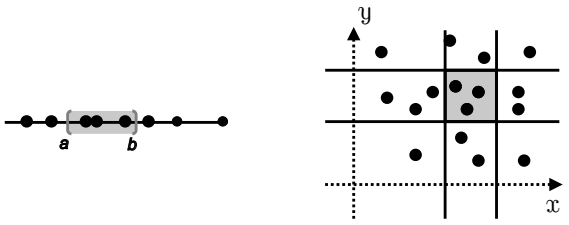
Рад пружа комплетан опис структуре и истражује ефикасне начине њене имплементације у програмском језику C++. Као део рада развијена је и имплементација каква тренутно не постоји, или није јавно доступна[1] – динамичко стабло опсега са операцијама ажурирања које се гради над потпуно произвољним типовима података. У раду се разматрају и методи побољшања перформанси на вишејезгарним процесорима. На крају је дата анализа резултата добијених над практичним подацима.

Као основна литература за поглавља 3.1, 3.2, 3.3 и 3.4 коришћене су [1], [10] и [2], док се поглавље 3.6 заснива на [4] и [5].

2 Ортогонална претрага опсега

У општем случају, претрага по опсегу се састоји од претпроцесирања скупа објеката S како би се утврдило који објекти из S се секу са упитним објектом који се назива *опсег*. Постоји више варијанти овог проблема, зависно од типа објеката који се траже (тачке, линије, полигони...), типа претраге (пријављивање, бројање, провера постојања објеката у опсегу...), типа опсега (правоугаоник паралелан координатним осама, круг...), итд.

Ми се у раду ограничавамо на претрагу d-димензионих тачака по d–димензионом опсегу који је ортогоналан, односно паралелан са координатним осама. У једнодимензионом случају траже се тачке на правој које леже у датом интервалу, у две димензије тачке у равни које припадају правоугаонику паралелном с координатним осама, у тродимензионом простору – тачке унутар ортогоналног квадра. У општем случају, задатак претраге је да пронађе све d-димензионе тачке које леже унутар ортогоналног d-квадра, односно векторског производа d ортогоналних интервала. Оваква претрага се назива „правоугаона“ претрага опсега, или, у рачунарској геометрији, *ортогонална претрага опсега*. Занима нас само случај у ком се над истим подацима извршавају вишеструки упити.



Слика 1: Ортогонална претрага опсега у једнодимензионом и дводимензионом случају.

Многи упити над базом се могу интерпретирати геометријски. Ако слогове у бази по-

[1] колико је аутору рада познато

сматрамо као тачке у вишедимензионом простору, упите над подацима из базе можемо превести у упите над скупом тачака. На пример, једна претрага опсега над доменом књига које су представљене тачкама с координатама *cena*, *str* и *god* би могла да враћа све књиге са ценом између 500 и 3000 динара, бројем страна између 100 и 300, и годином издавања између 1990. и 2010. год.

2.1 Једнодимензиона претрага опсега

Посматрамо прво једнодимензиони случај. Улаз је скуп тачака у једнодимензионом простору, нпр. у скупу реалних бројева $\mathbb{R}$, а траже се све тачке унутар интервала на реалној правој.

Проблем. За задати скуп тачака $S = \{p_1, p_2, \ldots, p_n\}$ на реалној правој и задати интервал $I = [x', x'']$ пријавити све тачке из S које се налазе у I.

2.1.1 Сортирани низ

Проблем се једноставно решава помоћу сортираног низа тачака и две варијанте бинарне претраге које налазе доњу, односно горњу границу интервала (слика 2). Ако кључ који траже не постоји у низу, ови методи враћају његовог следбеника, односно претходника.

Алгоритам BP_donja_granica(A, *Levi*, *Desni*, x)
Улаз. A (низ бројева уређених неопадајуће), *Levi*, *Desni* (опсег индекса претраге) и x (број који се тражи)
Излаз. *Poz* (најмањи индекс i такав да је $A[i] \geq x$, или 0 ако такав индекс не постоји)

```
1  while Levi < Desni:
2      Srednji = ⌊(Levi + Desni)/2⌋          // ⌈(Levi + Desni)/2⌉
3      if A[Srednji] > x:                    // <
4          Desni = Srednji − 1               // Levi = Srednji + 1
5      else:
6          Levi = Srednji                    // Desni = Srednji
7  if Levi = Desni and A[Levi] ≤ x:          // ≥
8      Poz = Levi
9  else:
10     Poz = 0
```

Слика 2: Налажење првог елемента који је већи или једнак (доња граница), односно последњег елемента који је мањи или једнак (горња граница) од тражене вредности. Кôд за налажење горње границе је због сличности дат у виду коментара у другој колони.

Једно решење је да се најпре претрагом доње границе x' у низу пронађе прва тачка која припада интервалу, а затим да се проласком кроз низ редом пријављују све тачке које иза ње следе, а и даље леже у интервалу. Временска сложеност бинарне претраге је $O(\log n)$, а пријављивање k тачака се обавља у $O(k)$ корака па је укупна сложеност претраге $O(\log n + k)$. Ово је пример алгоритма *осетљивог на излаз* (eng. output-sensitive)

чија сложеност зависи од броја резултата, односно величине излаза, а не само од величине улаза. Конструкција се ради само једном, и за њу је због сортирања потребно $O(n \log n)$ корака. Просторна сложеност је $O(n)$.

Наведени алгоритам приликом пријављивања тачака додатно врши k поређења са десним крајем интервала x''. Боље решење које минимизује број поређења је да, уместо једне, извршимо две бинарне претраге по оба краја интервала (и даље у $O(\log n)$ корака), и онда без поређења вредности пријавимо k елемената који се налазе између два добијена гранична индекса претраге. Ово ограничава број поређења вредности на највише $O(\log n)$, мада је убрзање само за константни фактор и не мења асимптотску сложеност алгоритма. Ипак, у пракси ово може бити од значаја, поготово ако су вредности у низу сложени типови података и њихово поређење је нетривијално.

Алгоритам Niz_pretraga_opsega(A, n, x', x'')
Улаз. A (низ величине n, сортиран неопадајуће), $[x', x'']$ (интервал)
Излаз. F (скуп тачака из A које припадају интервалу)

```
1   i = BP_donja_granica(A, 1, n, x')
2   {i је најмањи индекс за који је A[i] ≥ x' или 0}
3
4   j = BP_gornja_granica(A, 1, n, x'')
5   {j је највећи индекс за који је A[j] ≤ x'' или 0}
6
7   if i ≠ 0:
8       while i ≤ j:
9           додај A[i] у F
10          i = i + 1
```

Слика 3: Бинарна претрага опсега у сортираном низу.

Мана сортираног низа је што није могућа његова генерализација на више димензије, нити се низ може ефикасно ажурирати (уклањање елемента је обично скупа операција).

Теорема 2.1 *Скуп од n тачака на реалној правој може се претпроцесирати за време $O(n \log n)$ у сортирани низ величине $O(n)$, који тражи и пријављује све тачке које леже у датом интервалу у времену $O(\log n + k)$, где је k број пријављених резултата.*

2.1.2 Бинарно стабло претраге

Још једно ефикасно решење добија се коришћењем бинарног стабла претраге (БСП) које чува тачке (слика 4). Чвор стабла је слог који садржи кључ и показиваче на левог и десног сина ($kljuc$, $levi$, $desni$).

Претрага се изводи тражењем крајева интервала и спуштањем низ стабло све до чвора поделе у ком се путеви P_1 и P_2 ка крајевима интервала раздвајају (чвор 35 на слици 5). Након што се пронађе овај чвор, претрага се наставља и лево и десно, поредећи чворове стабла са крајевима интервала. При поређењу са левим крајем интервала, када претрага скреће улево, пријављују се сви чворови у десном подстаблу, и обрнуто, при поређењу са десним крајем интервала, када претрага скреће удесно, пријављују се сви чворови у левом

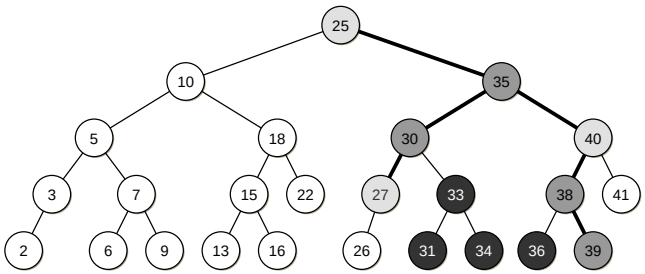
Алгоритам BSP_pretraga_opsega(v, x', x'')
Улаз. v (показивач на корен БСП), $[x', x'']$ (интервал)
Излаз. F (скуп тачака из v које припадају интервалу)

```
1  if v ≠ nil:
2      if x' ≤ v.kljuc:
3          BSP_pretraga_opsega(v.levi, x', x'')
4      if x' ≤ v.kljuc and x'' ≥ v.kljuc:
5          додај v.kljuc у F
6      if x'' > v.kljuc:
7          BSP_pretraga_opsega(v.desni, x', x'')
```

Слика 4: Претрага опсега у бинарном стаблу претраге.

подстаблу. Овим су обухваћени сви чворови у унутрашњим подстаблима која су „ограђена“ путевима P_1 и P_2, њих укупно k, где је k број пријављених резултата. Током обиласка, такође се проверавају и пријављују сви чворови на путевима који леже у интервалу.

Приметимо да се испод чвора поделе претрага практично „шири“ улево док год је леви крај интервала још увек лево од текућег чвора, и удесно док год је десни крај интервала још увек десно од текућег чвора. Овако се повећава опсег чворова за које се проверава припадност интервалу и прате само перспективне гране, односно оне које имају шансу да садрже неку тачку у интервалу, док све друге гране бивају одсечене.



Слика 5: Претрага интервала $[30, 39]$ у бинарном стаблу претраге. Путеви P_1 и P_2 су означени подебљаном линијом, гранични чворови сивом бојом (тамнијом ако улазе у резултат), а чворови које заклапају P_1 и P_2 црном бојом. Резултат чине тамно сиви и црни чворови.

Путеви не морају нужно да се завршавају у листовима (нпр. леви пут на слици 5 се завршава у чвору 27). Чворови дуж путева могу али не морају да припадају интервалу, ово је потребно додатно проверити (нпр. гранични чворови 25, 27 и 40 на слици 5 не припадају интервалу).

Ако је стабло уравнотежено, претрага посећује $O(\log n)$ граничних чворова и $O(k)$ „ограђених“ чворова, док спољашњи чворови бивају одсечени. Резултат чини унија свих ограђених чворова и оних чворова из P_1 и P_2 који припадају интервалу. Укупна сложеност претраге је $O(\log n + k)$.

Предност стабла у односу на сортирани низ је што је БСП динамичка структура и омогућава измене података након конструкције. Просторна сложеност је и даље $O(n)$, али

уз нешто већи константни фактор у односу на низ.

Теорема 2.2 *Скуп од n тачака на реалној правој може се претпроцесирати за време $O(n \log n)$ у уравнотежено бинарно стабло претраге величине $O(n)$, које тражи и пријављује све тачке које леже у датом интервалу у времену $O(\log n + k)$, где је k број пријављених резултата.*

2.2 Више димензије

Сортирани низ и БСП су врло добре структуре података за решавање проблема једнодимензионе ортогоналне претраге опсега, али нису погодне за случај када је број димензија већи од један. Међутим, ипак се могу користити као делови сложенијих структура које описујемо у наставку.

3 Стабло опсега

Стабло опсега (eng. range tree) је структура података која се гради над скупом d-димензионих тачака и ефикасно проналази све тачке које припадају неком ортогоналном d–димензионом опсегу. Ради једноставности, у наставку претпостављамо да су тачке јединствене по вредностима свих својих координата по свакој координатној оси, а у поглављу 3.5 показујемо како се ово ограничење може отклонити. Посматрамо прво једнодимензиони случај.

3.1 Једнодимензионо стабло опсега

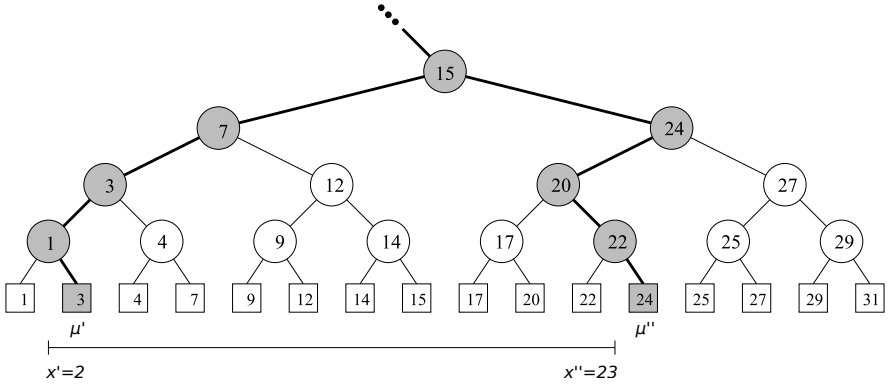
Једнодимензионо стабло опсега представља проширење уравнотеженог бинарног стабла претраге које смо већ видели из поглавља 2.1.2. Као такво, наслеђује све особине класичног БСП уз једну битну разлику – у стаблу опсега се подаци чувају сортирани у листовима. Ово омогућава да се стабло лако генерализује на вишедимензиони случај. Унутрашњи чворови служе искључиво за усмеравање претраге кроз стабло и практично деле простор претраге на дисјунктне области. Кључ унутрашњег чвора се може одредити произвољно све док раздваја чворове левог и десног подстабла по правилу БСП. По њему је кључ сваког чвора у стаблу већи или једнак од кључева свих чворова у његовом левом подстаблу, а строго мањи од кључева свих чворова у његовом десном подстаблу. Ми за вредност кључа унутрашњег чвора узимамо максимални кључ из његовог левог подстабла, односно крајњи десни лист. За запис чвора стабла користимо слог са следећим пољима:

- *levi* – показивач на лево подстабло или *nil* ако оно не постоји
- *desni* – показивач на десно подстабло или *nil* ако оно не постоји
- *velicina* – број листова у подстаблу чији је корен овај чвор (величина листа је 1)
- *kljuc* – вредност за усмеравање претраге кроз стабло (само за унутрашње чворове)
- *sek* – показивач на секундарно стабло у наредној димензији (само за унутрашње чворове у вишедимензионим стаблима)
- *tacka* – тачка, или веза ка тачки у неком облику (само за листове)

Подразумевамо да је овим слогом описана логичка структура сваког чвора, а за сада заменарујемо питања имплементације као што су да ли се два међусобно искључива поља попут *sek* и *tacka* могу чувати као једно поље, итд. Ове детаље дискутујемо у поглављу 3.8.

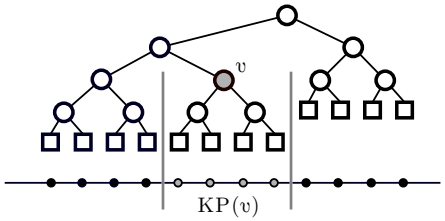
На слици 6 је приказано једно једнодимензионо стабло опсега. Њиме решавамо исти проблем из поглавља 2.1.

Проблем. За задати скуп тачака $S = \{p_1, p_2, \ldots, p_n\}$ на реалној правој и задати интервал $I = [x', x'']$, пријавити све тачке из S које се налазе у I.



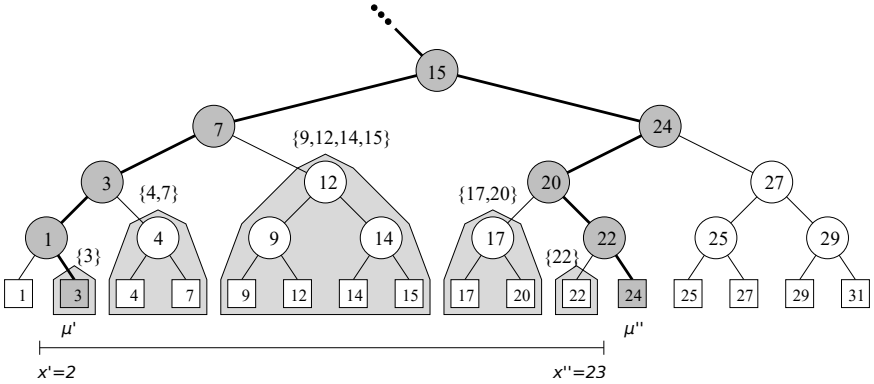
Слика 6: Претрага интервала $I = [2, 23]$ у 1D стаблу опсега.

Зарад поједностављења нотације, уводимо прво неке дефиниције. Нека је *канонички подскуп* чвора v, у ознаци $KP(v)$, скуп свих тачака из листова подстабла чији је он корен. Канонички подскуп корена стабла је, на пример, читав скуп тачака S. Чвор је *канонички* ако његов канонички подскуп припада траженом интервалу, а канонички подскуп његовог родитеља не. Канонички чворови су корени максималних подстабала која припадају интервалу претраге ([10]).



Слика 7: Канонички подскуп чвора v су тачке из листова његовог подстабла, на слици означене сивом бојом.

Због хијерархијске структуре стабла опсега, у примеру на слици 8 се лако види да су канонички подскупови $\{3\}, \{4, 7\}, \{9, 12, 14, 15\}, \{17, 20\}$ и $\{22\}$, који редом припадају каноничким чворовима $3, 4, 12, 17$ и 22, међусобно дисјунктни и да покривају интервал претраге. Другим речима, резултат претраге се увек састоји од уније дисјунктних каноничких подскупова каноничких чворова. Решавање проблема претраге се, дакле, своди на проналажење свих каноничких чворова и пријављивање њихових каноничких подскупова. Ако се путеви ка x' и x'' раздвајају у чвору v_p, није тешко приметити да су канонички чворови сви десни синови чворова на путу од v_p ка μ' када претрага скреће лево (чворови 12 и 4 на слици 8), сви леви синови чворова на путу од v_p ка μ'' када претрага скреће десно (чворови 17 и 22), и евентуално чворови у крајевима интервала, μ' и μ'' (чвор 3).



Слика 8: Претрага интервала $I = [2, 23]$ у 1D стаблу опсега. Изнад канонички чворова претраге $3, 4, 12, 17$ и 22, исписани су њихови канонички подскупови, а њихова подстабла су посебно оивичена и осенчена. Резултат претраге чини унија означених канонички подскупова.

Алгоритам претраге је дат на слици 10 и у целости је преузет из [1]. Да бисмо пријавили све тачке из S у интервалу I, поступамо на следећи начин. Најпре тражимо крајеве интервала x' и x'' низ стабло функцијом `Nadji_cvor_podele()` (слика 9) све до чвора у којем се путеви ка x' и x'' раздвајају. Ако се пре тога дође до листа, претрага се завршава и проверавамо да ли тачка у завршном листу евентуално припада интервалу. У супротном, пронађен је чвор поделе који представља најнижег заједничког претка два завршна чвора претраге, μ' и μ''. То је најнижи чвор у стаблу у чијим се подстаблима налазе све тачке у интервалу. Сада тражимо x' низ леву грану чвора поделе. Сваки пут када скренемо улево тражећи x', знамо да се тачке из каноничког подскупа десног сина тог чвора (синови 12 и 4 на слици 8) налазе у траженом интервалу. Пријављујемо их помоћу функције `Prijavi_kanonicki_podskup()` са слике 10. Слично, тражимо x'' низ десну грану чвора поделе. Сваки пут када скренемо удесно тражећи x'', знамо да се тачке из каноничког подскупа левог сина тог чвора (синови 17 и 22 на слици 8) налазе у траженом интервалу. Пријављујемо их истом функцијом. Овим су обухваћени сви чворови у унутрашњим подстаблима која су „ограђена“ путевима ка μ' и μ'', њих укупно k. Коначно, претрага завршава у листовима μ' и μ'' за које додатно проверавамо да ли припадају интервалу.

Алгоритам Nadji_cvor_podele(v, x', x'')
Улаз. v (показивач на корен стабла опсега), $[x', x'']$ (интервал)
Излаз. v (чвор поделе у ком се путеви до x' и x'' раздвајају,
или лист у ком се оба пута завршавају)

```
1  while v није лист:
2      if x'' ≤ v.kljuc:
3          v = v.levi
4      elif x' > v.kljuc:
5          v = v.desni
6      else:
7          break
```

Слика 9: Налажење чвора поделе у стаблу опсега.

Алгоритам 1D_stablo_opsega_pretraga(v, x', x'')
Улаз. v (показивач на корен 1D стабла опсега), $[x', x'']$ (интервал)
Излаз. F (скуп тачака из v које припадају интервалу)

```
1  v_podele = Nadji_cvor_podele(v, x', x'') {слика 9}
2  if v_podele је лист:
3      ако тачка из v_podele припада интервалу, додај је у F
4  else:
5      {прати леви пут од v_podele до x' и пријављуј десна подстабла}
6      v = v_podele.levi
7      while v није лист:
8          if x' ≤ v.kljuc:
9              Prijavi_kanonicki_podskup(v.desni)
10             v = v.levi
11         else:
12             v = v.desni
13     ако тачка из листа v припада интервалу, додај је у F
14     {прати десни пут од v_podele до x'' и пријављуј лева подстабла}
15     v = v_podele.desni
16     while v није лист:
17         if x'' > v.kljuc:
18             Prijavi_kanonicki_podskup(v.levi)
19             v = v.desni
20         else:
21             v = v.levi
22     ако тачка из листа v припада интервалу, додај је у F
```

function Prijavi_kanonicki_podskup(v)

```
1  if v је лист:
2      додај тачку из листа v у F
3  else:
4      Prijavi_kanonicki_podskup(v.levi)
5      Prijavi_kanonicki_podskup(v.desni)
```


Слика 10: Налажење опсега у 1D стаблу опсега.

Сложеност. Пошто је стабло опсега у основи уравнотежено БСП, просторна сложеност му је $O(n)$, а време изградње $O(n \log n)$. Каноничких чворова претраге има највише $2 \cdot \log n$, односно време које се утроши на њихово проналажење је $O(\log n)$. За сваки канонички чвор обиђе се по једно подстабло и пријаве његове тачке. Како је број унутрашњих чворова сваког *пуног*[2] бинарног стабла мањи од броја његових листова, то значи да је време обиласка стабла пропорционално броју његових листова. Ако је укупан број тачака које се пријављују у свих $O(\log n)$ подстабала једнак k, онда је укупно време које се утроши на обилазак $O(\log n)$ подстабала линеарно по k. Укупно време претраге је $O(\log n + k)$, где је k број резултата претраге.

Теорема 3.1 *Скуп од n тачака на реалној правој може се претпроцесирати за време $O(n \log n)$ у 1D стабло опсега величине $O(n)$, које тражи и пријављује све тачке које леже у датом интервалу у времену $O(\log n + k)$, где је k број пријављених резултата.*

3.2 Дводимензионо стабло опсега

Дводимензионо стабло опсега чува тачке из $\mathbb{R}^2$ и решава следећи проблем.

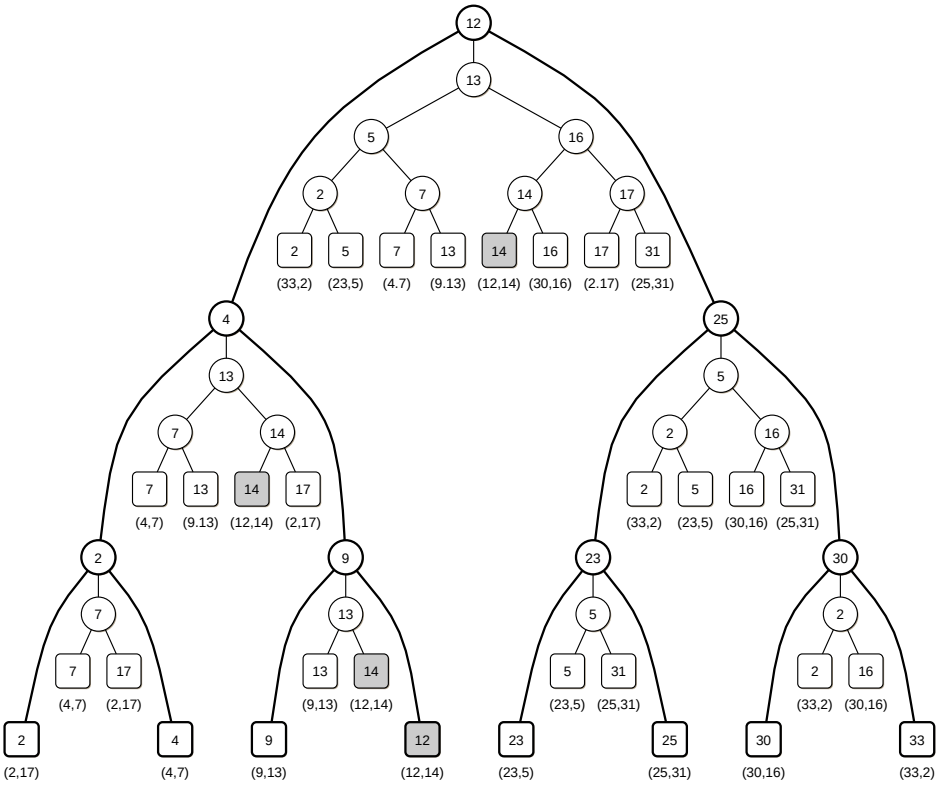
Проблем. За задати скуп тачака $S = \{p_1, p_2, \ldots, p_n\}$ у равни и задати интервал $I = [x', x''] \times [y', y'']$, пријавити све тачке из S које се налазе у I.

Да бисмо пронашли тачке унутар интервала I, можемо поступити на следећи начин. Прво алгоритмом за 1D претрагу пронађемо $O(\log n)$ каноничких чворова. За сваки такав чвор v знамо да у свом каноничком подскупу $KP(v)$ садржи све тачке чије x координате леже унутар интервала $[x', x'']$, али чије y координате нису обавезно у интервалу $[y', y'']$. Пошто 2D претрагу занима само онај њихов подскуп који лежи и у $[y', y'']$, уместо да тачке пријавимо, ми над њима извршимо још једну такву претрагу, али овај пут по њиховим y координатама и тако добијемо тачке које су у интервалу $[x', x''] \times [y', y'']$. Ово је још једна једнодимензиона претрага коју можемо да извршимо под условом да нам је доступно 1D стабло опсега по y координатама тачака из $KP(v)$. Управо на овоме се заснива идеја 2D стабла опсега:

- Главно стабло је 1D стабло опсега изграђено над x координатама тачака из S.
- Сваки унутрашњи чвор v у пољу $v.sek$ чува показивач на 1D стабло опсега изграђено над y координатама тачака из његовог каноничког подскупа $KP(v)$. Ова структура се назива *придружена структура* чвора v, или његово *секундарно стабло* (стабло у наредној димензији).

На слици 11 приказан је пример 2D стабла опсега. Сваки унутрашњи чвор у главном стаблу има придружено секундарно стабло по y координати које је приказано у средини, између његовог левог и десног подстабла. Свака тачка се у стаблу појављује више пута (примера ради, сва појављивања тачке $(12, 14)$ су означена сивом бојом). Структура има „фрактални“ изглед.

[2]Бинарно стабло је пуно ако сваки унутрашњи чвор у стаблу има тачно два сина.



Слика 11: 2D стабло опсега за тачке $(2, 17)$, $(4, 7)$, $(9, 13)$, $(12, 14)$, $(23, 5)$, $(25, 31)$, $(30, 16)$ и $(33, 2)$.

Алгоритам претраге 2D стабла је готово идентичан претрази 1D стабла осим једног детаља. Уместо да позивамо функцију за пријављивање тачака `Prijavi_kanonicki_podskup(`v`)` (линије 9 и 18 на слици 10), ми сада за сваки пронађени канонички чвор v приликом претраге позивамо функцију `1D_stablo_opsega_pretraga(`$v.sek$`, `y'`, `y''`)`. Другим речима, алгоритмом претраге за 1D стабло, ми у секундарном y-стаблу које је придружено чвору v тражимо и пријављујемо тачке из $KP(v)$ које по својој y координати припадају интервалу $[y', y'']$. Због сличности не наводимо алгоритам претраге за 2D стабло.

Алгоритам 2D претраге у сваком каноничком чвору главног стабла v проводи $O(\log n + k_v)$ време за претрагу по y координати, где је k_v број тачака које секундарно стабло чвора v пријави. Укупно време проведено у свим каноничким чворовима главног стабла (њих $O(\log n)$) је $\sum_v O(\log n + k_v)$. Како је $\sum_v k_v = k$, где је k укупан број пријављених тачака, а $\sum_v O(\log n) = O(\log^2 n)$, укупна сложеност 2D претраге је $O(\log^2 n + k)$.

Конструкција. На слици 12 је дат алгоритам за конструкцију 2D стабла опсега. На почетку се сортира улазни низ тачака по x и y координатама и добијају се нови низови S_x и S_y. Стабло даље градимо рекурзивно од ова два низа, одозго надоле. Циљ нам је да у сваком тренутку скуп тачака у оба низа буде исти, само сортиран по различитим координатама. Базни случај је ако имамо само једну тачку, када правимо лист који чува ту тачку. У супротном, у низу S_x по x координати тражимо медијану x_m и од ње правимо нови чвор v са кључем S_x. Од тачака из низа S_y градимо једнодимензионо стабло опсега по

y координати $\mathcal{T}_y$, које затим придружујемо као секундарно стабло чвору v. Медијаном x_m сада делимо тачке из S_x у два подскупа – S_x^{levi} у ком су тачке са x координатом мањом или једнаком од x_m, и S_x^{desni} у ком су тачке са x координатом већом од x_m. За изградњу левог и десног подстабла чвора v недостају нам само још еквивалентни скупови тачака сортирани по y координати, S_y^{levi} и S_y^{desni}. Њих добијамо из низа S_y тако што редом ископирамо све елементе који су по x координати лево, односно десно од x_m. Копирање чува релативни поредак тачака по y координати, односно резултатујући низови остају сортирани по y, а садрже исте подскупове тачака као S_x^{levi} и S_x^{levi}. За ово је довољно користити један помоћни низ величине $\lceil S_y/2 \rceil$. На крају, синове чвора v градимо рекурзивно на основу добијених x и y низова. Пошто поделу улаза радимо по средишњој тачки, коначан резултат је савршено уравнотежено стабло.

Приметимо да због особине алгоритма претраге да се увек креће граничним путевима по „ободу“ интервала и пријављује унутрашња подстабла, нека секундарна стабла никад неће бити употребљена у претрази. То је случај са секундарним стаблима свих чворова на ободу стабла (левој и десној ивици, укључујући корен стабла), и њих можемо да прескочимо приликом конструкције стабла. У рекурзивном алгоритму на слици 12 ова измена би могла да се састоји у увођењу два додатна параметра *LevaIvica* и *DesnaIvica* у функцију `Napravi_2D_stablo_opsega()` којима се у првом позиву функције на почетку алгоритма прослеђује вредност 1. Кад год се функција рекурзивно позове да направи подстабло левог, односно десног сина, параметру *DesnaIvica*, односно *LevaIvica* у позиву додељујемо вредност 0. На крају, у свим чворовима који нису ни на једној од ивица (оба параметра су 0) прескачемо изградњу секундарног стабла. Ова уштеда, иако корисна у пракси, ипак не мења асимптотску просторну сложеност стабла.

Алгоритам Konstrukcija_2D_stabla_opsega(S, n)
Улаз. S (низ тачака у равни дужине n)
Излаз. v (показивач на корен 2D стабла опсега)

```
1  if S = ∅: return nil
2  S_x = S сортиран неопадајуће по x коорд. тачака
3  S_y = S сортиран неопадајуће по y коорд. тачака
4  v = Napravi_2D_stablo_opsega(1, n, S_y)
```

function Napravi_2D_stablo_opsega($Levi$, $Desni$, P_y)

```
1  if Levi = Desni:
2      v.tacka = S_x[Levi]                          {остала поља листа v су nil}
3  else:
4      Srednji = ⌊(Levi + Desni)/2⌋
5      v.kljuc = S_x[Srednji].x
6      v.sek = направи 1D стабло опсега Ty по y коорд. тачака из P_y
7      Y = {p_i ∈ P_y | p_i.x ≤ v.kljuc}
8      v.levi = Napravi_2D_stablo_opsega(Levi, Srednji, Y)
9      Y = {p_i ∈ P_y | p_i.x > v.kljuc}
10     v.desni = Napravi_2D_stablo_opsega(Srednji + 1, Desni, Y)
11 return v
```

Слика 12: Конструкција 2D стабла опсега.

Сложеност. Почетна два сортирања по x и y координатама трају $O(n \log n)$ корака. У рекурзивној функцији, секундарно стабло се, као и свако друго БСП, од сортираног низа гради у $O(n)$ корака одоздо нагоре. Прављење нових y низова је такође линеарна операција. Рекурзивна функција дели улаз на два једнака или приближно једнака дела (ако му је дужина непарна), а затим од оба рекурзивно гради нова подстабла. Време које се проводи у сваком чвору главног стабла је линеарно величини улаза, односно његовог каноничког подскупа. Диференцна једначина за сложеност рекурзивне функције (без почетног сортирања) је $T(n) < 2 \cdot T(n/2) + O(n)$, $T(1) = 1$, а њено решење је $O(n \log n)$. Сортирање има исту сложеност, те је укупно време за конструкцију 2D стабла опсега $O(n \log n)$.

Просторна сложеност се може разматрати на два начина. Посматрано по нивоима, на сваком нивоу стабла од њих $O(\log n)$, свака тачка је смештена тачно једном, па је збирна величина свих секундарних стабала на једном нивоу $O(n)$. Укупна просторна сложеност је $O(n \log n)$. Посматрано по тачкама, у 2D стаблу опсега на слици 11 сивом бојом су означена сва појављивања једне од тачака у стаблу, њих $O(\log n)$. Видимо да се тачка налази у секундарним стаблима свих чворова дуж путање од тачке до корена стабла, односно тачно једном на сваком нивоу стабла. Пошто тачака има n, а нивоа $O(\log n)$, укупна просторна сложеност 2D стабла опсега је $O(n \log n)$. Број унутрашњих чворова у свих стаблима је линеаран броју листова ($n - 1$ јер су стабла пуна) па не мења ову сложеност.

Теорема 3.2 *Скуп од n тачака у равни може се претпроцесирати за време $O(n \log n)$ у 2D стабло опсега величине $O(n \log n)$, које тражи и пријављује све тачке које леже у датом правоугаонику паралелном с координатним осама у времену $O(\log^2 n + k)$, где је k број пријављених резултата.*

3.3 Више димензије

d–димензионо стабло опсега чува тачке из $\mathbb{R}^d$ и решава следећи проблем.

Проблем. За задати скуп тачака $S = \{p_1, p_2, \ldots, p_n\}$ из $\mathbb{R}^d$ и задати интервал $I = [x'_1, x''_1] \times [x'_2, x''_2] \times \ldots [x'_d, x''_d]$, пријавити све тачке из S које се налазе у I.

Стабло опсега може лако да се генерализује на случај већег броја димензија. По првој координати тачака из S конструишемо 1D стабло опсега, при чему сваком унутрашњем чвору v придружимо секундарно стабло димензије $d-1$, изграђено над тачкама из каноничког подскупа $KP(v)$, ограниченим на њихових последњих $d-1$ координата. Ово стабло опсега димензије $d-1$ се рекурзивно конструише на исти начин, као 1D стабло опсега по другој координати тачака, при чему сваки унутрашњи чвор v има придружено секундарно стабло димензије $d-2$, изграђено над тачкама из каноничког подскупа $KP(v)$, ограниченим на њихових последњих $d-2$ координата. Рекурзија се зауставља кад нам остану тачке ограничене на своју последњу координату, када градимо обично 1D стабло без секундарних структура.

Претрага функционише по сличном принципу као за 2D стабла. У стаблу по првој димензији проналазимо $O(\log n)$ каноничких чворова чији канонички подскупови садрже тачке чије су прве координате унутар траженог интервала. Даље се претражују њихове секундарне структуре по наредној димензији, и тако редом. У свакој димензији проналазимо $O(\log n)$ каноничких чворова. Рекурзија се зауставља у последњој димензији када пријављујемо резултате претраге. Нека је $T_d(n)$ време проведено у претрази d-димензионог стабла са n тачака, не рачунајући пријављивање тачака. Претрага d-димензионог стабла укључује претрагу главног стабла која траје $O(\log n)$, и претраживање логаритамског броја стабала димензије $d-1$. Диференцна једначина за сложеност претраге је $T_d(n) < O(\log n) + O(\log n) \cdot T_{d-1}(n)$, $T_2(n) = O(\log^2 n)$, а њено решење је $O(\log^d n)$. Укупна сложеност претраге d-димензионог стабла је $O(\log^d n + k)$, где је k број пријављених тачака.

Конструкција. Ради комплетности, дајемо и приказ алгоритма за конструкцију d-димензионог стабла опсега, међутим овај пут описујемо нешто другачије решење које није проста генерализација претходног алгоритма за конструкцију 2D стабла. Овим алгоритмом се може изградити стабло опсега произвољне димензије, па и димензија 1 и 2 која смо видели до сад. Улаз је низ тачака из $\mathbb{R}^d$ које на почетку алгоритма сортирамо неопадајуће по њиховој првој координати. Стабло градимо рекурзивно, одозго надоле, слично као у 2D алгоритму. Новина је параметар i који означава димензију стабла коју тренутно конструишемо. Низ тачака се стандардно дели на две половине (у $O(1)$ пошто је сортиран) и од њих се рекурзивно граде лево и десно подстабло. Затим, под условом да нисмо стигли до последње димензије у којој чворови немају придружена секундарна стабла, конструишемо секундарно стабло за текући чвор v. За ово су нам потребне тачке из каноничког подскупа чвора v, сортиране по $(i+1)$-ој координати. Канонички подскуп чвора v се састоји од уније каноничких поскупова његовог левог и десног подстабла. Пошто су лево и десно подстабло чвора v већ у потпуности рекурзивно изграђени по свим преосталим димензијама, тачке које су нам потребне можемо у линеарном времену добити из њихових секундарних стабала. За ово користимо функцију `Uzmi_sortirane_sek_tacke(u)` са слике 13 која инфиксно обилази секундарно стабло $u.sek$ и тако из његових листова прикупља тачке у редоследу сортираном неопадајуће по $(i+1)$-ој координати. У случају кад је подстабло u лист, враћа се његова (сортирана) тачка. Затим се тачке добијене из два подстабла у линеарном времену обједињују у један низ по $(i+1)$-ој координати процедуром за обједињавање која је део

стандардног алгоритма за сортирање обједињавањем. Коначно, од овако спојених тачака гради се секундарно стабло и придружује текућем чвору v. Приметимо да се у алгоритму секундарна стабла граде одоздо нагоре. Сваки чвор практично обједињује сортиране листе тачака своје деце у линеарном времену, почев од листова који су тривијално сортирани. Ово је у суштини исти алгоритам као сортирање обједињавањем.

Сложеност. Почетно сортирање по првој координати траје $O(n \log n)$. Њега изузимамо из анализе рекурзивне функције. Нека је $T_d(n)$ време потребно за конструкцију d-димензионог стабла опсега од n тачака. Рекурзивна функција за једнодимензионо стабло у сваком чвору проводи константно време за комбиновање резултата потпроблема, па се 1D стабло гради у $T_1(n) = O(n)$, јер је почетни низ био сортиран. Конструкција d-димензионог стабла се састоји од прављења БСП од сортираних тачака, што траје $O(n)$, и прављења свих придружених секундарних структура. У главном стаблу по првој димензији, у чворовима на сваком нивоу стабла, свака тачка стабла се чува у тачно једној придруженој секундарној структури. Укупно време потребно да се изграде све придружене структуре (димензије $d-1$) свих чворова на неком нивоу стабла је $O(T_{d-1}(n))$, колико треба да се изгради придружена структура корена. Диференцна једначина за сложеност рекурзивне функције (без почетног сортирања) је $T_d(n) < O(n) + O(\log n) \cdot T_{d-1}(n)$, $T_1(n) = O(n)$, а њено решење је $O(n \log^{d-1} n)$. Почетно сортирање не утиче на ову сложеност, те је ово и укупно време за конструкцију d-димензионог стабла опсега. Просторна сложеност се изводи потпуно аналогно.

Алгоритам Konstrukcija_kD_stabla_opsega(S, n, k)
Улаз. k (број димензија стабла), S (низ тачака из $\mathbb{R}^k$ дужине n)
Излаз. v (показивач на корен k-D стабла опсега)

```
1  if S = ∅: return nil
2  сортирај S неопадајуће по првој координати тачака
3  v = Napravi_kD_stablo_opsega(S, 1, n, 1)
```

function Napravi_kD_stablo_opsega(X**,** $Levi$**,** $Desni$**,** i**)**

```
1  if Levi = Desni:
2      v.tacka = X[Levi]                          {остала поља листа v су nil}
3  else:
4      Srednji = ⌊(Levi + Desni)/2⌋
5      v.kljuc = X[Srednji].x_i
6      v.levi = Napravi_kD_stablo_opsega(X, Levi, Srednji, i)
7      v.desni = Napravi_kD_stablo_opsega(X, Srednji + 1, Desni, i)
8      if i < k:
9          P_levi = Uzmi_sortirane_sek_tacke(v.levi)
10         P_desni = Uzmi_sortirane_sek_tacke(v.desni)
11         обједини P_levi и P_desni у низ тачака сорт. по i + 1-ој коорд. P
12         P_n = Levi − Desni + 1 {величина низа P}
13         v.sek = Napravi_kD_stablo_opsega(P, 1, P_n, i + 1)
14 return v
```

function Uzmi_sortirane_sek_tacke(v**)**

```
1  if v је лист:
2      убаци v.tacka у T
3  else:
4      инфиксно обићи секундарно стабло v.sek и за
           сваки посећени лист u, убаци u.tacka у T
5  return T
```

Слика 13: Конструкција k-димензионог стабла опсега.

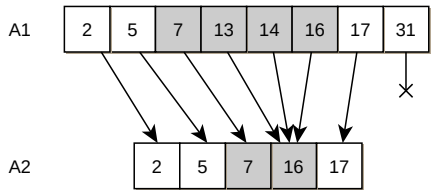
Теорема 3.3 *Скуп од n тачака из $\mathbb{R}^d$ ($d \geq 2$) може се претпроцесирати за време $O(n \log^{d-1} n)$ у d-димензионо стабло опсега величине $O(n \log^{d-1} n)$, које тражи и пријављује све тачке које леже у датом опсегу у времену $O(\log^d n + k)$, где је k број пријављених резултата.*

3.4 Унакрсно повезивање

Унакрсно повезивање (eng. cross-linking) је техника која убрзава претрагу у стаблу опсега за логаритамски фактор. Део је шире технике која се зове *делимична пропагација* (eng. fractional cascading), а која служи да убрза низ бинарних претрага по истом кључу у низу уређених структура. Делимична пропагација је у општем случају дефинисана над графовима, а назив је добила по томе што се делови низова структура са доњих нивоа подижу (пропагирају) нагоре, у друге низове, како би се избегли неки патолошки случајеви приликом претраге. У нашем случају за овим нема потребе јер због особина стабла до граничних случајева не долази. Из тог разлога, ми у наставку користимо само један део њене основне идеје, део о повезивању структура, а изостављамо онај по ком је техника добила име, па је зовемо само – унакрсно повезивање. У литератури се, пак, ова техника примењена над стаблима опсега често помиње као делимична пропагација.

Применом унакрсног повезивања претрага опсега у 2D стаблу може да се обави у $O(\log n + k)$. Стабло опсега прво налази скуп тачака које леже у интервалу $[x', x'']$ као унију $O(\log n)$ каноничких подскупова. Сваки канонички подскуп се даље претражује по интервалу $[y', y'']$ у придруженом секундарном стаблу по y координати. Пошто се у сваком секундарном стаблу током претраге тражи *исти* кључ y', можемо да побољшамо време претраге у $O(1)$ по стаблу. Идеја је да на почетку алгоритма урадимо претрагу у првом секундарном стаблу у $O(\log n)$ времену, али да онда на неки начин искористимо ту информацију како бисмо ефикасније претражили остале структуре. Кључ је успоставити „паметне“ везе између структура изграђених за канонички подскупове које желимо да претражимо.

Посматрајмо прво једноставнији пример (слика 14).



Слика 14: Тражење интервала $[6, 16]$ у повезаним низовима.

Имамо два низа бројева бројева сортираних неопадајуће, A_1 и A_2, где је $A_2 \subset A_1$. За дати интервал $[y', y'']$ желимо да пријавимо све кључеве из A_1 и A_2 који леже у интервалу. Позивањем функције `Niz_pretraga_opsega()` (слика 3), по једном за сваки низ, решење добијамо у $O(\log n + k)$ корака, где је k укупан број пронађених резултата у обе претраге.

Међутим, пошто је $A_2 \subset A_1$, може се приступити паметније. Додајемо показиваче између елемената из A_1 и елемената из A_2 тако да $A_1[i]$ чува показивач на први елемент из A_2 чији је кључ већи или једнак од кључа $A_1[i]$. Ако сада желимо да урадимо претрагу по интервалу $[y', y'']$, довољно је да извршимо само претрагу у A_1 по y' и редом пријављујемо елементе који следе све док су у интервалу. За ово је потребно $O(\log n + k_1)$ корака. Ако се претрага у A_1 завршила у $A[i]$, користимо његов показивач да започнемо претрагу у A_2. Ово се ради у константном времену, а претрага у A_2 траје $O(1 + k_2)$ корака. На крају, унакрсно повезивање нам омогућава да претражимо два низа са истим кључем у $O(\log n + k)$, где је k укупан број резултата обе претраге.

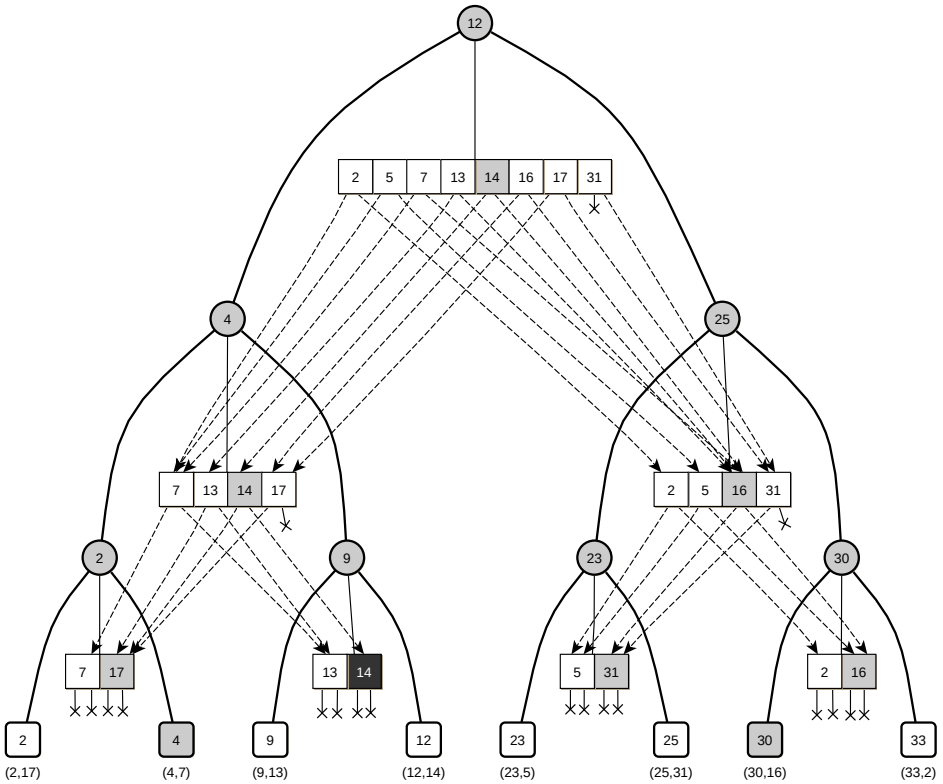
Ово се једноставно примењује на 2D стабло опсега. Кључно запажање је да у стаблу

важи да је канонички подскуп сваког чвора надскуп каноничког подскупа сваког од његових синова, и да практично имамо идентичан случај као у примеру на слици 14.

Стабло по x координати остаје исто, али уместо изградње y-стабала за каноничке подскупове, смештамо их као сортиране низове по y координати. Поље $v.sek$ у свим унутрашњим чворовима сада постаје показивач на секундарни низ слогова са следећим пољима:

- *levi* – показивач на први елемент у низу $v.levi.sek$ чија је y координата тачке већа или једнака од y координате тачке $v.sek.tacka$, или *nil* ако такав елемент не постоји у $v.levi.sek$
- *desni* – показивач на први елемент у низу $v.desni.sek$ чија је y координата тачке већа или једнака од y координате тачке $v.sek.tacka$, или *nil* ако такав елемент не постоји у $v.desni.sek$
- *tacka* – показивач на тачку

Резултат трансформације 2D стабла опсега са слике 11 у стабло са унакрсно повезаним низовима приказан је на слици 15. Добијено стабло се назива *слојевито стабло опсега* (eng. layered range tree).



Слика 15: Слојевито 2D стабло опсега. Скуп тачака је исти као на слици 11. Истакнути су резултати претраге интервала $I = [3, 28] \times [14, 17]$.

Претрага слојевитог стабла опсега се одвија на следећи начин. Прво уобичајеном методом у главном стаблу проналазимо чвор поделе. Затим у секундарном низу чвора поделе вршимо претрагу по левом крају интервала y' методом $BP\_donja\_granica()$ са слике 2, прилагођеном нашем низу слогова. Тако у $O(\log n)$ добијамо индекс i првог елемента у низу $v_{podele}.sek$ чија се тачка налази у границама интервала по y координати, или 0 ако такав индекс не постоји. Ако индекс не постоји, то значи да у главном стаблу нема тачака у опсегу и тиме завршавамо претрагу. У противном, нека је показивач на пронађени елемент низа $P = v_{podele}.sek[i]$. Настављамо даље да обилазимо главно стабло уобичајеним алгоритмом, тражећи крајеве интервала. Међутим, у сваком кораку кретања сада истовремено напредујемо и показивач P и истом смеру у коме се креће главна претрага (нпр. ако главна претрага скрене улево, и P скрећемо улево да показује на леви низ испод). Ако у било ком тренутку P постане nil, претрага у смеру тог краја интервала се прекида као неуспешна. Када пронађемо подстабло које желимо да претражимо по y координати, ми у P већ имамо одговор где по y координати у том подстаблу евентуално почињу елементи у опсегу. Довољно је проћи кроз низ почев од P, и редом пријављивати све тачке које леже у опсегу. Ово се обавља у $O(1 + k_v)$ уместо досадашњих $O(\log n + k_v)$ корака, где је k_v број тачака у подстаблу чији је корен v.

Техником унакрсног повезивања се сложеност претраге 2D стабла опсега умањује за један логаритамски фактор и постаје $O(\log n + k)$. Цену плаћамо додатним простором за показиваче, али он је константан и не мења укупну просторну сложеност од $O(n \log n)$. За d-димензиона стабла ($d \geq 2$) довољно је користити слојевито 2D стабло у претпоследњој димензији и сложеност претраге се аутоматски смањује на $O(log^{d-1} n + k)$.

Још једна варијанта слојевитог стабла уместо једног, чува два пара показивача по елементу низа – један пар за доњу границу, а други за горњу границу. Ово убрзава претрагу за константни фактор али је главна предност што омогућава још једну примену стабла опсега – брзо бројање елемената у опсегу у $O(1)$, насупрот итерације и $O(k_v)$ упоређивања која се не може избећи кад се користи само један пар показивача.

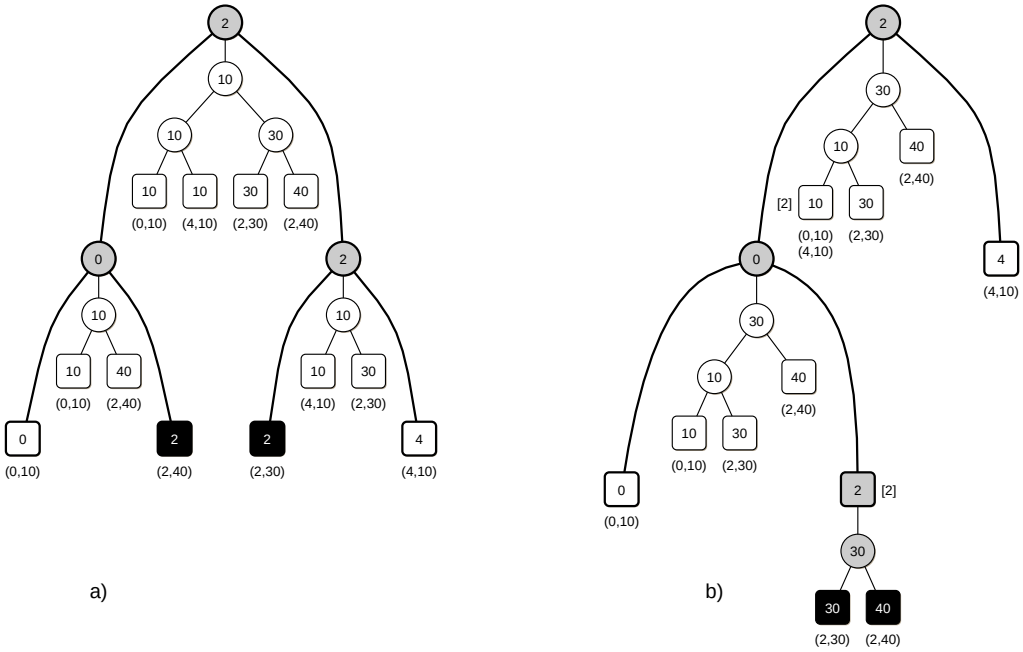
Унакрсно повезане низове није једноставно одржавати након операција које мењају стабло, па се обично користе само у статичким стаблима. Ипак, алгоритми постоје и за динамичка стабла, али су доста сложенији.

Теорема 3.4 *Скуп од n тачака из $\mathbb{R}^d$ може се претпроцесирати за време $O(n \log^{d-1} n)$ у слојевито d-димензионо стабло опсега величине $O(n \log^{d-1} n)$, које тражи и пријављује све тачке које леже у датом опсегу у времену $O(\log^{d-1} n + k)$, где је k број пријављених резултата.*

3.5 Обрада дупликата у стаблу

Ради једноставности, до сад смо претпостављали да све тачке у стаблу имају јединствене координате по свакој од димензија стабла, што у пракси често није случај. У наставку приказујемо како се ово ограничење може отклонити.

Могуће су разне стратегије како би се подржале тачке са идентичним појединачним координатама, или чак тачке идентичне по свим координатама. На слици 16 су приказане две такве шеме.



Слика 16: Две могуће шеме смештања тачака $(0, 10)$, $(2, 40)$, $(2, 30)$ и $(4, 10)$ у 2D стаблу опсега, при чему неке тачке имају идентичне појединачне координате. Истакнути су резултати претраге интервала $I = [1, 2] \times [0, 50]$.

Прва шема (слика 16а) чува дупликате као засебне чворове у свакој димензији стабла. Да би претходно описани алгоритам претраге (слика 10) проналазио све дупликате у опсегу, потребно је прво изменити метод за проналажење чвора поделе (слика 9). На слици прве шеме видимо да се дупликати могу налазити са обе стране чвора поделе као што је то случај са чворовима са вредношћу 2 у првој димензији стабла. Зато је сада, приликом тражења чвора поделе, потребно стати чим се пронађе први чвор који је у опсегу, а не ићи лево кад је испуњена једнакост у линији 2 на слици 9. Ово се постиже променом знака $\leq$ у $<$ у наведеној линији. Такође, испод чвора поделе, приликом обиласка десне путање до μ'' опет треба узети у обзир евентуално постојање дупликата у левом подстаблу, односно у линији 17 на слици 10 променити знак $>$ у $\geq$. У случајевима када дупликата у стварности с десне стране нема (нпр. у секундарном стаблу корена главног стабла, два чвора са вредношћу 10 се налазе са леве стране чвора поделе), тада се десна путања обилази почев од нешто више позиције у стаблу него што је заиста неопходно. Срећом, ово се дешава само у случају кад у

стаблу постоји тачка са координатом која је једнака левом крају интервала претраге. Ипак, ово не мења укупну сложеност претраге јер спуштање од било ког чвора до листа у стаблу траје највише $O(\log n)$, што је мање од $O(\log n + k)$, а омогућава нам да на једноставан начин подржимо тачке са једнаким координатама.

Друга шема (слика 16b) чува дупликате у једном чвору, али уз додатно секундарно стабло по наредној димензији у којој се координате овако груписаних тачака могу разликовати (нпр. лист 2 у главном стаблу садржи секундарно стабло за две тачке). У последњем нивоу стабла тачке се чувају у листи (нпр. лист 10 у секундарном стаблу корена главног стабла садржи тачке $(0, 10)$ и $(4, 10)$). За идентичне тачке могу се користити одговарајући бројачи. Ако доста тачака дели исту координату, оваква стабла су компактнија и претрага је бржа јер налази мање каноничких чворова. Ова шема је згоднија и за операцију брисања тачака из стабла коју описујемо у поглављу 3.6.3, јер се идентичне тачке, под условом да их имплементација стабла дозвољава, увек налазе груписане у једном листу.

3.6 Динамичко стабло

Стабло опсега је примарно замишљено као статичка структура података у којој се подаци након конструкције не могу мењати, па смо се до сада фокусирали на статичку верзију проблема ортогоналне претраге. У овом поглављу описујемо шта је потребно како би се у стабло могле накнадно убацивати нове или брисати постојеће тачке, а да притом буде потпуно очувана ефикасност претраге.

Алгоритам за конструкцију стабла опсега са слике 13 прави савршено уравнотежено стабло у ком се висине левог и десног подстабла у сваком чвору разликују највише за један. Овим се осигурава да висина стабла, а уједно и сложеност претраге није већа од $O(\log n)$. Међутим, уметање нових или брисање постојећих тачака из стабла после неког времена може да наруши његову уређеност. Да би се ово спречило, уместо обичних БСП користе се самоуравнотежујућа стабла претраге која аутоматски одржавају своју висину малом тако што реструктуирају стабло након што се равнотежа поремети. Постоји више начина да се ово постигне. *Локална реконструкција* или *уравнотежавање* је техника која за ово користи једну или више ротација чворова, *делимична реконструкција* је алтернативна техника која реконструише читава подстабла када постану неуравнотежена, а *глобална реконструкција* периодично реконструише целу структуру ([6]).

Постоје разне врсте самоуравнотежујућих стабала као што су АВЛ или црвено-црна стабла у којима се висине левих и десних подстабала чворова разликују за адитивни, односно мултипликативни фактор. Ова стабла припадају класи висински уравнотежених стабала. Поред њих, постоји и класа тежински уравнотежених стабала у којима равнотежа сваког чвора зависи од *тежине* његових подстабала. Значење појма „тежина“ овде није стриктно одређено – за њега се може узети нпр. број чворова у подстаблу, број листова у подстаблу, или било која друга погодна дефиниција. Ми у наставку, ако није другачије наглашено, дефинишемо функцију $tezina(v)$ као број листова у подстаблу чији је корен чвор v. Сваки чвор чува тежину подстабла чији је он корен, и тежине левог и десног подстабла се у сваком тренутку разликују за неки константан фактор.

3.6.1 $BB[\alpha]$ стабло

У наставку, као основу за динамичко стабло опсега бирамо најстарије и најједноставније тежински уравнотежено стабло: *стабло ограничене равнотеже* (eng. tree of Bounded Balance) са параметром α, или краће – $BB[\alpha]$ стабло. Стабла ограничене равнотеже се разликују од осталих класа БСП у томе што имају параметар који може да варира, а којим се бира компромис између дужине трајања претраге и учестаности реструктуирања стабла. Ова стабла су једноставна за одржавање и поред операција убацивања и брисања чворова, а време претраге је тек нешто дуже него у савршено уравнотеженим стаблима.

Тежински уравнотежена стабла имају механизме да очувају тежину левог и десног подстабла сваког чвора приближно истом. Формално, за сваки чвор v $BB[\alpha]$ стабла ($\alpha < \frac{1}{2}$) важи:

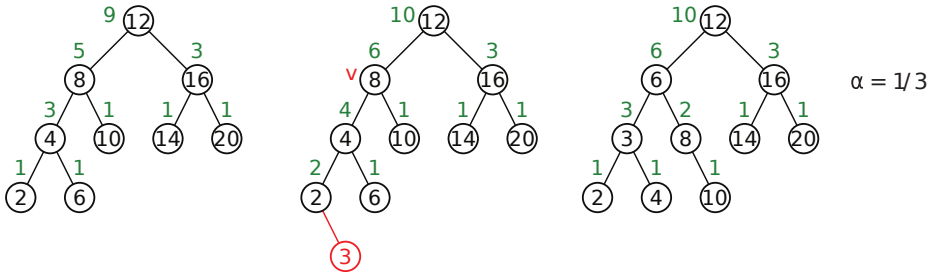
$$tezina(v.levi) \geq \lfloor \alpha \cdot tezina(v) \rfloor$$

$$tezina(v.desni) \geq \lfloor \alpha \cdot tezina(v) \rfloor$$

Треба обратити пажњу на одабир параметра α јер, на пример, за $\alpha = \frac{1}{2}$, стабло мора да буде савршено уравнотежено у сваком тренутку. Много је боље ако одаберемо мање α, нпр.

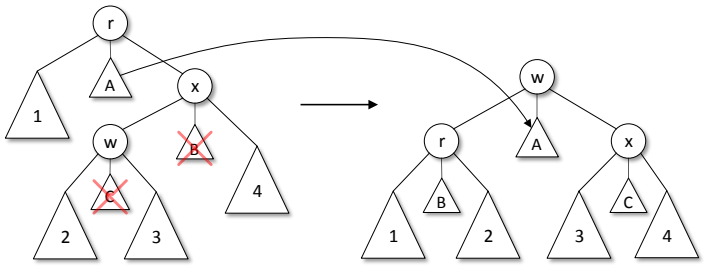
$\alpha = \frac{1}{5}$. Тежинска уравнотеженост је јаче својство од висинске уравнотежености – тежински уравнотежено стабло има висину највише $\log_{\frac{1}{1-\alpha}} n$ ([5]).

Слика 17 приказује једно $BB[\alpha]$ стабло пре, у току, и после уметања чвора 3 који нарушава равнотежу у чвору v ($1 < 6 \cdot \frac{1}{3}$). Подстабло са кореном у чвору v се затим реконструкцише и резултат је поново уравнотежено стабло по дефиницији $BB[\alpha]$ стабла. За вредност функције $tezina(v)$ у овом примеру је одабран број чворова у подстаблу v, а тежине су приказане уз чворове.



Слика 17: Уметање елемента у $BB[\alpha]$ стабло и реконструкција подстабла у највишем чвору чији је фактор равнотеже нарушен.

Међутим, у [7] се показује да уравнотежавање стабла опсега ротацијама није у тој мери ефикасно као у обичним БСП јер сложеност ротације више није $O(1)$ пошто доводи до поновне изградње секундардних стабала у вишим димензијама. На слици 18, без улажења у детаље самог поступка ротације, дајемо приказ вишедимензионог стабла опсега пре и после уравнотежавања двоструком ротацијом и компликација до којих она доводи. Са слике видимо да, иако се на крају добија уравнотежено стабло, два од три секундарна стабла (B и C) морају у целости да се реконструишу јер им је ротација променила канонички подскуп чворова. Пре ротације секундарно стабло B је било изграђено над тачкама из скупа подстабала $\{2, 3, 4\}$, а након ротације B би требало да садржи тачке из подстабала $\{1, 2\}$; слично важи и за стабло C.



Слика 18: Двострука ротација у kD стаблу опсега и компликације које узрокује. Секундарна стабла B и C морају да се изграде испочетка.

Како овај корак реконструкције у ротацијама доминира, корисно је минимизовати њихову учестаност. То се једноставно постиже употребом поменуте технике делимичне реконструкције у комбинацији са нижом вредношћу параметра α, као алтернативе уравнотежавању помоћу ротација. Показује се да ако имамо просторну сложеност и време изградње

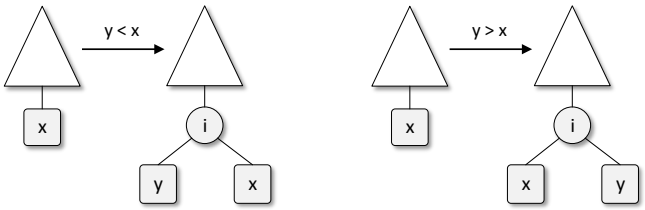
стабла опсега од $O(n \log^{d-1} n)$, да коришћењем амортизације и технике делимичне реконструкције лако можемо да учинимо претходно статичка стабла динамичким.

Ажурирања у класичном $BB[\alpha]$ стаблу које користи технику делимичне реконструкције могу бити веома брза. Обично кад додамо или обришемо неки чвор из стабла, то утиче само на околне чворове и обавља се у $O(1)$. Посебно, у стаблу опсега тај чвор је увек лист (јер су тачке у листовима), што је лакши случај. Повремено, већи део стабла постаје неуравнотежен, и тад можемо да уништимо читав тај део стабла и изградимо га испочетка. Кад то урадимо, конструишемо га као савршено уравнотежено подстабло. Након тога у стању смо да извршимо најмање $\Theta(k)$ нових операција уметања или брисања пре него што исто постабло опет постане неуравнотежено, где је k величина подстабла ([4]). Када опет треба да реконструишемо то подстабло, можемо да „наплатимо“ процес реконструисања од $\Theta(k)$ операција ажурирања које смо у међувремену „бесплатно“ извели. Тиме добијамо амортизовано време ажурирања од $O(1)$: делећи време реконструкције подстабла од $\Theta(k)$ са бројем $\Theta(k)$ операција ажурирања које нас практично нису ништа коштале, добијамо да нас је у просеку свака операција коштала $O(1)$. Овде претпостављамо да се подстабло може реконструисати у $\Theta(k)$ корака, на начин како је описано у поглављу 3.6.2. Међутим, како свака тачка коју додамо може потенцијално да избаци из равнотеже сва секундарна стабла од корена до листа у којима се налази (њих укупно $O(\log n)$), ту губимо један логаритамски фактор, односно можемо да ажурирамо стабло тек у амортизованом времену $O(\log n)$. За вишедимензиона стабла на исти начин добијамо амортизовано ажурирање у $O(\log^d n)$ корака, а и даље имамо време претраге $O(\log^d n + k)$ (за динамичка стабла не користимо технику унакрсног повезивања). У наставку описујемо тачан поступак уметања и брисања, без понављања сложености тих операција.

Теорема 3.5 *Скуп од n тачака из $\mathbb{R}^d$ ($d \geq 2$) може се претпроцесирати за време $O(n \log^{d-1} n)$ у d-димензионо стабло опсега величине $O(n \log^{d-1} n)$, које тражи и пријављује све тачке које леже у датом опсегу у времену $O(\log^d n + k)$, где је k број пријављених резултата, а врши уметање и брисање тачака у амортизованом времену $O(\log^d n)$.*

3.6.2 Уметање

Претпоставимо да желимо да уметнемо тачку p у вишедимензионо стабло опсега. Поступамо на следећи начин. Тражимо p у главном стаблу по њеној првој координати како бисмо утврдили њену позицију међу листовима и успут убацујемо p у сва секундарна стабла на која наиђемо дуж путање до листа. За једнодимензиона секундарна стабла користимо стандардни начин уметања чворова описан испод, иначе рекурзивно користимо исту процедуру. Приликом претраге, у сваком чвору v проверавамо да ли би евентуално убацивање тачке p у његово подстабло у коме би претрага иначе наставила, нарушило равнотежу чвора v. Ако је то случај, није потребно да настављамо даље, пронашли смо највиши чвор у стаблу ког операција уметања избацује из равнотеже. Тада реконструишемо читаво подстабло са кореном у чвору v тако што га обиђемо инфиксно, прикупимо у низ A тачке из његових листова које су због инфиксног обиласка сортиране по текућој димензији, и изградимо ново стабло над тачкама из $A \cup p$. Ново стабло је савршено уравнотежено стабло опсега у коме се број листова у левом и десном подстаблу сваког унутрашњег чвора разликује за највише један.



Слика 19: Уметање елемента y у стабло опсега.

У противном, ако нема чвора са нарушеном равнотежом, сваком чвору v дуж пута унапред повећавамо тежину, и претрага на крају завршава у листу са вредношћу x (слика 19). Нека је y вредност координате тачке p у текућој димензији. Правимо нови унутрашњи чвор i и њиме замењујемо чвор x, а x спуштамо на место његовог десног, односно левог сина, зависно од тога да ли је y мање или веће од x. Коначно, уписујемо y на одговарајућу позицију као брата чвора x и, ако нисмо у последњој димензији, правимо секундарно стабло за чвор i.

3.6.3 Брисање

У стаблу проналазимо лист са тачком коју желимо да обришемо. Симетрично уметању, брисање резултира уклањањем два чвора из стабла – листа x и његовог родитеља i који бива замењен својим другим сином y (слика 19). Ако при рекурзивном повратку од обрисаног чвора до корена стабла наиђемо на чвор коме је нарушена равнотежа, реконструишемо читаво подстабло на начин описан у поглављу 3.6.2. У супротном, бришемо тачку из секундарних структура свих унутрашњих чворова које посетимо на путу до корена.

3.7 Примене

Поред своје основне намене за проналажење и пријављивање тачака у опсегу, стабло опсега може да извршава и још неке упите који се ефикасније имплементирају као засебне операције.

3.7.1 Бројање тачака у опсегу

Ако се тражи *број тачака у опсегу* (eng. range counting query), а не саме тачке, можемо приступити на следећи начин. У сваком чвору стабла опсега изграђеног по његовој последњој димензији чувамо величину ($v.velicina$) каноничког подскупа тог чвора, односно број листова који се налазе испод њега. Измена претраге се односи само на последњу димензију стабла. Како се спуштамо низ стабло тражећи границе интервала μ' и μ'' у два $O(\log n)$ обиласка, сабирамо величине свих подстабала која улазе у одговор, а која бисмо иначе пријавили.

Испоставља се да у експлицитном динамичком стаблу чворови већ садрже овај податак и да га интерно користе за тежинско уравнотежавање стабла, па га не морамо посебно чувати. Имплицитна статичка стабла због своје особине да су пуна и *комплетна*[3] могу да израчунају број листова у $O(1)$ по формули $\lceil n/2 \rceil$, где је n дужина низа, па се број тачке у последњој димензији и код њих броје у константном времену.

Да би се избегла сложеност од $O(\log^{d-1} n + k)$ у слојевитим стаблима из поглавља 3.4, неопходно је користити варијанту са 4 показивача. Пре претраге стабла у претпоследњој димензији, вршимо две бинарне претраге низа по границама интервала у последњој димензији. За ово користимо методе `BP_donja_granica()` и `BP_gornja_granica()` (слика 2) и тако добијамо граничне тачке опсега у корену стабла у последњој димензији. Затим пролазимо кроз стабло у претпоследњој димензији и у сваком кораку истовремено мењамо вредности показивача на граничне тачке из наредне димензије тако да прате кретање низ стабло у истом смеру (ако претрага у претпоследњој димензији скрене улево, односно удесно, показивач узима вредност свог левог, односно сина, спуштајући се тако један ниво ниже у стаблу). Овако у константном времену налазимо границе опсега у свим каноничким чворовима и на коначан број тачака додајемо разлику добијених индекса. Време извршавања постаје $O(\log^{d-1} n)$, тј. више не зависи од k.

Укупна сложеност операције бројања тачака у опсегу за остале варијанте стабла је $O(\log^d n)$ за стабло димензије d, а просторна сложеност се не мења ни за константни фактор.

3.7.2 Провера постојања бар једне тачке у опсегу

Утврђивање да ли бар једна тачка припада опсегу је још једноставнији проблем. У стаблу у последњој димензији тражимо леви крај интервала, проверавајући притом у сваком кораку да ли је текући чвор у опсегу. Ако наиђемо на такав чвор, одмах одговарамо позитивно и алгоритам прекидамо пре спуштања до листа. Претрага иначе завршава у листу μ' ког проверавамо и дајемо коначан одговор. Сложеност операције је $O(\log^d n)$ за стабло димензије d. У слојевитим стаблима све ово радимо у претпоследњој димензији, и у сваком кораку додатно проверавамо да ли је показивач на повезани низ из последње

[3] Бинарно стабло је комплетно ако су сви нивои стабла скроз попуњени осим можда последњег, а на последњем су чворови поравнати слева.

димензије различит од nil, у ком случају враћамо позитиван одговор пре силаска до листа. Сложеност операције за слојевита d-стабла је $O(\log^{d-1} n)$.

3.7.3 Најтежа тачка у опсегу

Под претпоставком да тачке у 1D стаблу имају придружене тежине, може се у $O(\log n)$ корака пронаћи тачка са највећом тежином у опсегу. Сваки чвор може да чува најтежу тачку у свом каноничком подстаблу. Поступамо слично као при бројању тачака у опсегу, само што уместо сабирања величина подстабала каноничких чворова дуж леве и десне путање, овај пут тражимо максимум њихових тежина. У динамичком стаблу се након убацивања или брисања чворова ове информације лако ажурирају при рекурзивном повратку од измењеног чвора до корена стабла. Сложеност за вишедимензиона и слојевита стабла се изводи слично као у осталим применама.

3.8 Организација података у меморији

Тачке је у неком облику неопходно чувати само у последњој димензији стабла јер се пријављивање нормално обавља само на том нивоу. Међутим, пошто претрага вишедимензионог стабла у свакој димензији k мора да провери да ли су гранични листови μ' и μ'' у опсегу по k-ој и преосталим димензијама, да бисмо ово утврдили имамо две могућности. Можемо одабрати да листови чувају тачке у свим димензијама стабла (или бар везе ка тачкама у неком облику) и онда извршити директну проверу, или, алтернативно, можемо да и за листове градимо секундарна стабла у свим димензијама (од једног чвора) која онда рекурзивно претражујемо док не стигнемо до последњег који чува тачку. Одлучујемо се за прво решење јер је брже и заузима мање простора.

У пракси, подаци у стаблу не морају бити геометријски већ тачке могу бити сложени типови података из произвољног, али тотално уређеног домена. Уз то, тачке могу имати и високу меморијску захтевност и компликоване операције копирања и поређења. У том случају је згодно да се тачке у меморији чувају на једном месту (нпр. у низу за статичка, или у двоструко повезаној листи за динамичка стабла), а да се у стаблима, уместо копија вредности, чувају само индекси, односно показивачи на тачке. С друге стране, у статичким стаблима представљеним имплицитно преко низова, чување копија кључева у унутрашњим чворовима може знатно да побољша ефикасност кеш меморије процесора јер се при поређењу кључева у току обиласка стабла не морају пратити показивачи на спољашње локације које су потенцијално изван кеш меморије.

Такође, често је корисно да тачке поред координата над којима се гради стабло, додатно имају и придружену вредност произвољног типа. Она може бити део тачке (допунска координата која не учествује у изградњи стабла), или се ова веза може чувати на неки други начин.

4 Програмска реализација

Дате су имплементације статичког и динамичког стабла опсега у програмском језику C++, у виду шаблонских класа `RangeTree` и `DynamicRangeTree`. Пошто им је интерна репрезентација потпуно различита и постоји разлика у интерфејсу, имплементиране су као две засебне класе. Прва користи имплицитно представљање стабла помоћу низа и сортирани низ у последњој димензији, а друга имплементира тежински уравнотежено $BB[\alpha]$ стабло са делимичном реконструкцијом при операцијама ажурирања. Обе класе долазе у заглављу *RangeTree.h* које је довољно само укључити у пројекат, без додатних зависности приликом превођења и повезивања програма. Превођење је могуће на свим системима за које постоји C++ компајлер који подржава стандард C++14.

Подаци у стаблу не морају нужно бити геометријски (тачке из $\mathbb{R}^k$) као што је у претходном излагању подразумевано, већ тип података за сваку димензију може бити из произвољног домена све док су испуњени следећи услови:

- тип има дефинисан конструктор копије
- тип има дефинисан `operator<`

Додатно, ако се жели подршка за испис, потребно је да је за тип сваке координате тачке дефинисан оператор исписа на излазни ток (`operator<<`). Параметризација типова се врши помоћу стандардног C++ механизма – шаблона. Класе су моделоване да прате дизајн и филозофију стандардне C++ библиотеке (STL) и једноставне су за коришћење. Следећи пример показује како се конструише и користи основно једнодимензионо стабло опсега:

```
1  using namespace rt;
2
3  // конструишемо статичко стабло на основу тачака
4  RangeTree<int> rangeTree({ 23, 37, 80, 97 /* ... */ });
5
6  // налазимо тачке у интервалу I = [17, 65]
7  // добијамо назад std::vector<RangeTree<int>::Point>
8  auto result = rangeTree.pointsInRange({ 17, 65 });
9
10 // сортирамо и исписујемо добијене резултате
11 std::sort(result.begin(), result.end());
12 for (const auto& pt : result)
13     std::cout << pt << std::endl;
```

Резултат претраге не мора бити уписан у колекцију предефинисаног типа (`std::vector`) као у претходном примеру, већ је могуће проследити излазни итератор произвољног типа. Наредни пример гради дводимензионо стабло и за излаз користи итератор који при упису убацује тачке на крај двоструко повезане листе, као и итератор који директно исписује резултујуће тачке на стандардни излаз, без складиштења у меморију:

```cpp
using namespace rt;
using RT = RangeTree<double, double>;

// иницијализујемо генератор случајних бројева
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(-10.0, 10.0);

// динамички попуњавамо низ тачака
std::vector<RT::Point> points;
for (int i = 0; i < 1000000; i++)
    points.emplace_back(dis(gen), dis(gen));

RT rangeTree{points};   // или std::move(points)

// дефинишемо интервал претраге I = (2.5, 6.72] × [−3.145, +∞]
RT::Range range{ { open(2.5), 6.72 }, { -3.145, +inf } };

// пунимо двоструко повезану листу са највише 10 тачака у опсегу
std::list<RT::Point> result;
rangeTree.pointsInRange(range, std::back_inserter(result), 10);

// претрага директно исписује тачке у опсегу на стандардни излаз
auto outIt = std::ostream_iterator<RT::Point>(std::cout, "\n");
rangeTree.pointsInRange(range, outIt);
```

Савет за побољшање брзине претраге је да се типови димензија стабла при декларацији наводе оним редом који потенцијално може да што раније елиминише највећи број тачака, јер се претрага изводи редом од прве до последње димензије, сужавајући у свакој димензији скуп тачака кандидата.

Претрага се може вршити над отвореним, полуотвореним, и затвореним интервалима, а границе интервала могу бити једнаке и бесконачности ($\pm\infty$) која је универзална за све типове података. Димензионалност интервала претраге одговара броју димензија стабла. Интервали се у кôду записују на следећи начин:

```cpp
    { a, b }                        // [a, b]
    { open(a), closed(b) }          // (a, b]
    { a, open(b) }                  // [a, b)
    { a, +inf }                     // [a, +∞)
    { -inf, b }                     // (−∞, b]
    { -inf, +inf }                  // (−∞, +∞)
    all                             // (−∞, +∞)
    { { a, b }, { c, d }, ... }     // [a, b] × [c, d] × ...
```

Границе интервала су подразумевано затворене.

Коначно, улазне тачке стабла опсега могу имати и придружену додатну вредност произвољног типа. Такво стабло се декларише на нешто другачији начин. Следећи пример уједно демонстрира и операције динамичког стабла:

```
using namespace rt;

using RT = DynamicRangeTree<Dimensions<double, double>, Value<const City *>>;

// стабло је иницијално празно
RT rangeTree;

// пунимо га подацима из базе
for (const auto& city : db.cities())
    rangeTree.insert({ city.latitude(), city.longitude(), &city });

// тражимо градове по географској ширини и дужини, и сваки
// пронађени резултат штампамо директно, без копирања у колекцију
RT::Range range{ { 42.94, 44.17 }, { 19.62, 21.325 } };
rangeTree.forEachPointInRange(range, [](const RT::Point& pt) {
    const City *city = pt.value();
    std::cout << city->name() << ": " << city->population() << std::endl;
});
```

У пракси је често корисно имати придружене вредности. На пример, кад год се над неким скупом објеката жели извршити претрага опсега по неком *подскупу* њихових поља, згодно је тачкама које се убацују у стабло и учествују у претрази додатно придружити показивач на матични објекат да би се касније имао приступ свим пољима објеката резултата. Претходни пример тако гради стабло опсега на основу географских ширина и дужина градова на мапи, и свака тачка стабла додатно има придружен показивач на слог који описује град и садржи информације као што су његов назив, број становника, надморска висина, итд.

Како би вишедимензионо стабло опсега могло да се параметризује и изгради са различитим типовима података у различитим димензијама, у статичко типизираном језику какав је C++ потребно је доста шаблонског метапрограмирања. Примера ради, за смештање и приступ координатама тачака димензије n не може се користити једноставан низ дужине n или нека слична структура података јер је свака координата потенцијално различитог типа. Типови нису унапред познати већ их задаје корисник класе и може их бити неограничено много. Исто тако, стабла која увезујемо по свакој димензији такође могу бити различитог типа јер пореде координате различитог типа. У имплементацији су за ово коришћене могућности новијих ревизија C++ стандарда као што су шаблони са произвољним бројем аргумената, n-торке (`std::tuple`) које захтевају индексирање у фази превођења програма, итд. Све ово чини интерфејсе мање читљивим па је из тог разлога у поглављу 4.1 дат *упрошћен* јавни интерфејс релевантних C++ класа, а на неким местима чак и у форми псеудо-кôда.

4.1 Структура класа

```
#ifndef RANGE_TREE_H_
#define RANGE_TREE_H_

namespace rt {

// бесконачност није имплементирана као подкласа класе
// IntervalEndpoint<T> јер би у том случају њена синтакса
// била inf<double>{} или -inf<std::string>{} уместо ±inf
class inf_t {
public:
    inf_t();
    inf_t operator+() const;
    inf_t operator-() const;
    bool isPositive() const;
    bool isNegative() const;
};

class all_t {
public:
    all_t();
};

static const inf_t inf;
static const all_t all;

template<typename T>
class IntervalEndpoint {
public:
    IntervalEndpoint(T const& value, bool isOpen = false);
    IntervalEndpoint(const inf_t& inf);
    IntervalEndpoint(const IntervalEndpoint& other);
    IntervalEndpoint& operator=(const IntervalEndpoint& rhs);
    bool isOpen() const;
    bool operator<(T const& value) const;
    bool operator>(T const& value) const;
    bool operator<=(T const& value) const;
    bool operator>=(T const& value) const;
    ~IntervalEndpoint();
};

template<typename T> IntervalEndpoint<T> open(T const& value);
template<typename T> IntervalEndpoint<T> closed(T const& value);

class NoMappedValue;

template<Dimensions<typename... DimensionTypes>,
    Value<typename ValueType> = NoMappedValue>
class RangeTree {
```

```
49  public:
50      class SimplePoint {
51      public:
52          // конструше тачку на основу координата
53          SimplePoint(const DimensionTypes&... coordinates);
54
55          // враћа координату тачке за дату димензију; индекс мора
56          // бити константа позната у време превођења програма, није
57          // могуће користити промењиву
58          template<size_t Dim> const auto& get() const;
59
60          // Основни оператори поређења за тачке. Координате тачака се по
61          // димензијама пореде лексикографски, слева надесно. Први метод
62          // омогућава кориснику да нпр. сортира резултате претраге.
63          bool operator<(const SimplePoint& other) const;
64          bool operator==(const SimplePoint& other) const;
65          friend std::ostream& operator<<(std::ostream& os, const SimplePoint& pt);
66
67          virtual ~SimplePoint();
68      };
69
70      class MappedPoint : public SimplePoint {
71      public:
72          MappedPoint(const DimensionTypes&... coordinates, const ValueType& value);
73          const ValueType& value() const;
74          bool operator<(const MappedPoint& other) const;
75          bool operator==(const MappedPoint& other) const;
76          friend std::ostream& operator<<(std::ostream& os, const MappedPoint& pt);
77      }
78
79      // дефинише тип тачке у зависности од тога да ли корисник захтева мапирање
80      using Point = typename std::conditional<std::is_same<ValueType,
81          NoMappedValue>::value, SimplePoint, MappedPoint>::type;
82
83      template<typename T>
84      class Interval {
85      public:
86          Interval(const IntervalEndpoint<T>& low, const IntervalEndpoint<T>& high);
87          Interval(const all_t&);
88          Interval(const Interval& other);
89          Interval& operator=(const Interval& rhs);
90          const IntervalEndpoint<T>& low() const;
91          const IntervalEndpoint<T>& high() const;
92          bool contains(T const& value) const;
93      };
94
95      class Range {
96      public:
97          // ако постоји само једна димензија, прослеђује параметре конструктору
98          // класе Interval и тако омогућава синтаксу { a, b } уместо { { a, b } }
99          template<typename... Args> Range(Args const&... args);
```

```
100         Range(const Interval<DimensionTypes>&... is);
101
102         template<size_t Dim = 0> const auto& get() const;
103         template<size_t DimBegin = 0, size_t DimEnd = sizeof...(DimensionTypes)>
104         bool contains(const Point& pt) const;
105     };
106
107     // конструктор који прихвата улаз из произвољне колекције преко итератора
108     template<typename InputIt> RangeTree(InputIt first, InputIt last);
109
110     // помоћни конструктор за често коришћени тип колекције
111     RangeTree(const std::vector<Point>& points);
112
113     // реконструше стабло на основу нових тачака
114     template<typename InputIt> assign(InputIt first, InputIt last);
115
116     // враћа број тачака у стаблу
117     size_t size() const;
118
119     // враћа све тачке у опсегу или њих највише n; општи метод који за излаз
120     // прихвата излазни итератор произвољног типа и у њега уписује резултате
121     template<typename OutputIt>
122     void pointsInRange(const Range& range, OutputIt outIt, size_t n = 0) const;
123
124     // помоћни метод за претрагу који враћа често коришћени тип колекције
125     std::vector<Point> pointsInRange(const Range& range, size_t n = 0) const;
126
127     // метод за итерацију кроз резултате претраге, без копирања у излазну колекцију
128     // омогућава брзи приступ интерним подацима у стаблу
129     // за сваки пронађени резултат позива се корисникова ф-ја
130     void forEachInRange(const Range& range, void (*f)(const Point& pt), size_t n = 0) const;
131
132     // враћа број тачака у опсегу
133     size_t countInRange(const Range& range) const;
134
135     // утврђује да ли постоји нека тачка у опсегу
136     bool existsInRange(const Range& range) const;
137
138     virtual ~RangeTree();
139 };
140
141 // разлика у интерфејсу у односу на класу RangeTree
142 // сви други методи и типови остају исти
143 template<Dimensions<typename... DimensionTypes>,
144     Value<typename ValueType> = NoMappedValue>
145 class DynamicRangeTree {
146 public:
147     // подразумевани конструктор који прави празно стабло
148     DynamicRangeTree(double alpha = 0.2);
149     // конструктор који прихвата улаз из произвољне колекције преко итератора
150     template<typename InputIt>
```

```
151     DynamicRangeTree(InputIt first, InputIt last, double alpha = 0.2);
152     DynamicRangeTree(const std::vector<Point>& points, double alpha = 0.2);
153
154     void insert(const Point& pt);
155     bool remove(const Point& pt);
156     bool empty() const;
157     void clear();
158 };
159
160 }       // namespace rt
161
162 #endif   // RANGE_TREE_H_
```

4.2 Резултати

4.2.1 Тестни подаци

У сврху тестирања коришћена је база података о свим градовима на свету са око 3,100,000 слогова, и њен подскуп, база свих места у Србији са око 15,000 слогова. Подаци се могу пронаћи и бесплатно преузети са интернет странице https://www.maxmind.com/en/free-world-cities-database. Грађена су 2D стабла на основу географских ширина и дужина градова, као и 3D стабла са бројем становника као додатном координатом. Поред географских података коришћене су и случајно генерисане тачке са униформном расподелом, у предефинисаном опсегу по свакој од координата. Поређења су рађена и са алгоритмом грубе силе који редом пролази кроз тачке и пријављује све оне које су у опсегу у времену $O(n)$.

4.2.2 Поређење различитих верзија структуре

Тестиране су две структуре: статичко стабло опсега представљено имплицитно преко низова, са сортираним низом уместо стабла у последњој димензији (није коришћена техника унакрсног повезивања), и динамичко стабло са показивачима. На рачунару који је коришћен за тестирање (Intel Core i5 3.4GHz), статичко стабло извршава око 50 упита у секунди над базом од 3,100,000 градова на мапи, са просечно 15% пријављених резултата. Ово је у случају кад се резултати претраге уписују у излазну колекцију, када корак уписа због алокације меморије доминира у односу на време претраге. Ако се уместо уписивања користи метод за претрагу `forEachInRange()`, где стабло позива корисникову функцију за сваку пронађену тачку без алоцирања меморије за резултате, онда структура над истом базом постиже око 1000 упита у секунди. Над базом градова у Србији стабло извршава око 30,000, односно 100,000 упита у секунди, зависно од тога да ли се резултати претраге уписују у колекцију или не. Упити су правоугаоници на мапи чије су координате биране као случајни бројеви са униформном расподелом из опсега по ширини, односно висини мапе. И поред тога што обе структуре раде веома брзо за практичне примене, занимљиво је да је динамичко стабло, иако савршено уравнотежено, за исте упите давало око 3.5 пута лошије резултате. Оволика разлика у брзини може се оправдати једино компактношћу низа и чињеницом да су његови елементи смештени на узастопним локацијама у меморији, што знатно побољшава ефикасност кеш меморије процесора (праћење показивача на локације изван кеш меморије је на данашњим процесорима скупа операција). Такође, још

једна разлика је што статичко стабло у последњој димензији резултате пријављује методом `Niz_pretraga_opsega()` из поглавља 3, што је потенцијално брже од обиласка подстабала каноничких чворова код динамичког стабла. Из тог разлога, испробано је повезивање листова на последњем нивоу динамичког стабла у повезану листу. Подстабло каноничког чвора се у том случају пријављује тако што се прво прочита број листова L у подстаблу (из поља $v.velicina$ у корену подстабла), а затим се спуштањем до крајњег левог листа и праћењем нововодатих показивача $v.sled$ пријављује L листова. Показало се да додавање повезане листе тек незнатно убрзава претрагу у просеку (за пар процената).

Затим је покушано да се проблему приступи на другачији начин. За алокацију чворова при изградњи динамичког стабла употребљен је посебан детерминистички меморијски алокатор са сложеношћу алокације од $O(1)$, који узастопно алоциране објекте истог типа смешта на узастопне локације у меморији. Повремено, кад се његов интерни блок од n објеката попуни, алокатор од система затражи нови, који је на другој меморијској локацији, али се величина блокова може одабрати да се ово не дешава често. Овим је постигнуто да чворови динамичког стабла буду на блиским меморијским локацијама баш као чворови у низовима статичког стабла, а да стабло притом задржи могућност ажурирања. Као резултат, ово је значајно убрзало и конструкцију и претрагу великих стабала. Претрага динамичког стабла је постала само око 2 пута спорија од статичког у просеку. Разлог овоме је вероватно тај што се динамичка стабла граде рекурзивно, одозго надоле, а након алокације сваког унутрашњег чвора, одмах се гради његово секундарно стабло у наредној димензији. Овим се добија погрешан редослед алокације чворова и распоред у меморији који није оптималан за претрагу. С друге стране, статичко стабло се гради одоздо нагоре, алоцирањем низа за све чворова стабла у текућој димензији одједном. Ако би се алгоритам конструкције динамичког стабла прилагодио особинама алокатора и променио да прво направи све чворове стабла у текућој димензији па тек онда њихова секундарна стабла, вероватно би се добио одговарајући распоред у меморији и жељено убрзање претраге (ово је остављено да се испроба у пракси).

4.2.3 Паралелизација

Даље је експериментално покушано убрзање конструкције и претраге стабла паралелизацијом. У недостатку вишепроцесорских система коришћене су технике конкурентног програмирања са нитима на процесору са 4 физичка језгра и истим бројем нити. Извршавање нити било је ограничено на унапред одабрана језгра, без могућности да их оперативни систем негде премести.

Паралелизовање процеса конструкције је било успешно, са ефикасношћу од 80%, пропорционално броју доступних језгара. За ово је измењен рекурзивни алгоритам изградње динамичког стабла да на одговарајућем нивоу стабла у току изградње свакој нити додели да рекурзивно конструише по једно подстабло. Ово се нпр. ради на другом нивоу стабла за две нити, на трећем нивоу стабла кад су доступне четири нити, односно четири процесорска језгра, итд. Конструкција стабла је готова кад све нити заврше са радом.

Један теоретски начин убрзања претраге за $\log n$ фактор, описан у [3], био би да се одвоји $O(\log n)$ процесора и сваком од њих додели по један ниво стабла у првој димензији, односно пријављивање највише два каноничка чвора колико их може бити на сваком нивоу стабла. Међутим, у пракси са нитима и стаблима мале висине, а велике димензионалности се показало да ово није изводљиво јер време потребно за комуникацију међу нитима превазилази

време потребно за претрагу. Симулирано је и претраживање на $O(\log n)$ нивоа са thread pool-ом и константним убацивањем нових задатака које би прва слободна нит преузела и извршила.

Ако у току претраге свака нит уписује своје резултате у заједнички излаз, то захтева синхронизацију приступа, где нити онда чекају свој ред да добију приступ за упис. У случају када је број резултата претраге превелик, показало се да нити више времена проводе чекајући једна другу на право за упис него радећи свој посао. Решење би наизглед могло да буде у томе да свака нит уписује резултате у сопствени бафер (који расте), па да се на крају резултати свих нити споје у један. То можда није добро за верзију претраге која прихвата излазни итератор ако није пожељно чување свих резултата у меморији ни у једном тренутку претраге, већ пријављивање „у лету“ (излазни итератор нпр. може да уписује на диск јер објекти резултати поред самог стабла не могу да стану у меморију). Овде је проблем већ техничке природе, а то је да би свака нит морала да има сопствени меморијски алокатор који нема исти проблем са закључавањем и синхронизацијом приступа (ово искључује могућност коришћења стандардних C++ механизама за алокацију) да до истог проблема са чекањем не би дошло.

На крају се увиђа да уопште није неопходно да се тачке уписују у излаз у тренутку кад је утврђено да се налазе у траженом интервалу, већ се могу запамтити само границе њиховог опсега у последњој димензији стабла (пар показивача на две тачке у завршном сортираном низу у статичким стаблима, и корен подстабла каноничког чвора у динамичким стаблима). Пошто стабло које стаје у радну меморију не може имати превелику висину, узима се да је висина од $O(\log n)$ за практичне примене константна (највише ширина процесорске речи), па се за излаз може унапред алоцирати бафер фиксне величине у који нити уписују *опсег* пронађених резултата, и то тако што свака нит добије свој део бафера за упис, па никаква синхронизација није потребна. На крају, довољно је на n нити тривијално разделити посао „проширивања“ елемената овог бафера у низ тачака резултата које се враћају кориснику.

За просте типове тачака експериментално се показало да се најбоље перформансе добијају када се простор претраге подели на не више од два дела (лево и десно подстабло у првој димензији) и у оба паралелно изврше претраге. Резултати се прикупе као показивачи или парови индекса у низ мало већи од $2 \cdot \log^{d-1} n$ и затим се са максималним бројем нити паралелизује попуњавање резултата у довољно велики излазни низ. За верзију претраге са излазним итератором довољно је из једне нити уписати све резултате. Тек ако је димензионалност стабла велика, или ако су стабла изграђена над сложеним типовима података са спорим операцијама поређења, коришћење већег броја нити има смисла и добија се приметно убрзање. Ово је испробано са тачкама чије су операције поређења биле симулиране тако да се извршавају дуже.

4.3 Могућа унапређења класа

Постоје разни начини на које би се имплементације класа могле даље побољшати. У наставку наводимо само неке од њих.

Кориснику класе би се могла дати могућност да параметризује објекат тачка, ког у тренутној имплементацији дефинише класа, произвољним типом за којег би сам имплементирао шаблонски „адаптер“ преко *одлика типова* (eng. type traits). Овако стабло не би морало да чува сопствену копију података, већ би се могло изградити над постојећим подацима из меморије (нпр. над низом показивача на објекте из базе) ако се класи на

одговарајући начин саопшти како да им приступа у току претраге. Ово је корисно јер је просторна сложеност стабла опсега таква да стабло већ са неколико милиона вишедимензионих тачака на данашњим рачунарима врло брзо превазилази количину доступне меморије у систему. У таквој имплементацији би стабло само „индексирало“ постојеће податке.

Згодно би било и да се као напредна опција при декларацији стабла, кориснику остави могућност да бира између неколико начина интерне организације података у стаблу путем тзв. *шаблонских „политика“* (eng. policy-based design). Оне би се комбиновале у фази превођења програма, и тиме не би утицале на брзину извршавања, а корисник би могао да сам одабере компромис између брзине извршавања претраге и заузећа меморије (нпр. да користи компактнију репрезентацију зарад мањег губитка на брзини). Такође би се класи могао дати наговештај да ли се очекује да тачке имају јединствене координате по једној или више координатних оса, односно димензија, да ли тачке са идентичним координатама могу имати сопствене или само заједничке придружене вредности и сл. Многе од оваквих параметара је пожељно знати у фази превођења како би шаблони били у стању да генеришу што бољи кôд. Сви ови избори утичу на крајњу организацију података у меморији, а пошто није практично имплементирати посебну верзију класе за сваку од њихових комбинација, композиција шаблонских политика је боље решење.

5 Закључак

У овом раду су приказани алгоритми за конструкцију, претраживање и ажурирање стабла опсега. Развијене су C++ имплементације статичке и динамичке верзије стабла које се могу изградити над произвољним типовима података. У поглављу 4.3 је наведено који су могући правци даљег развоја ове библиотеке класа.

Литература

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: *Computational Geometry: Algorithms and Applications*, Springer; 3rd edition; 2008

[2] Dinesh P. Mehta, Dinesh P. Mehta, Sartaj Sahni: *Handbook of Data Structures and Applications*; Chapman and Hall/CRC; 2004

[3] Pieter H. Hartel, Michiel H.M. Smid, Leen Torenvliet, Wilem G. Vree: *A Parallel Functional Implementation of Range Queries*

[4] Mark H. Overmars, Michiel H.M. Smid, Mark T. de Berg, Marc L. van Kreveld: *Maintaining range trees in secondary memory Part I: Partitions*; Acta Informatica (1990) 27:423

[5] Peter Brass: *Advanced Data Structures*; City College of New York; 2008

[6] Yi-Jen Chiang, Roberto Tamassia: *Dynamic Algorithms in Computational Geometry*; 1992

[7] Michael G. Lamoureux: *An Implementation of a Multidimensional Dynamic Range Tree Based on an AVL Tree*; 1995

[8] Миодраг Живковић: *Алгоритми*; Математички факултет, Београд; 2000

[9] Bjarne Stroustrup: *The C++ Programming Language, 4th Edition*; 2013

[10] Michelle M. Hugue: *Lecture 20: Range Trees*
https://www.cs.umd.edu/users/meesh/420/ContentBook/FormalNotes/MountNotes/lecture20-rangetrees.pdf