TDDD86 – Extralaboration #1

6 september 2019

I den här uppgiften får du göra en fullständig implementation av en typgenerisk klass för att lagra data i ett kd-träd. Filerna du behöver för att komma igång finns som `extralabb1.tar.gz` på kurshemsidan.

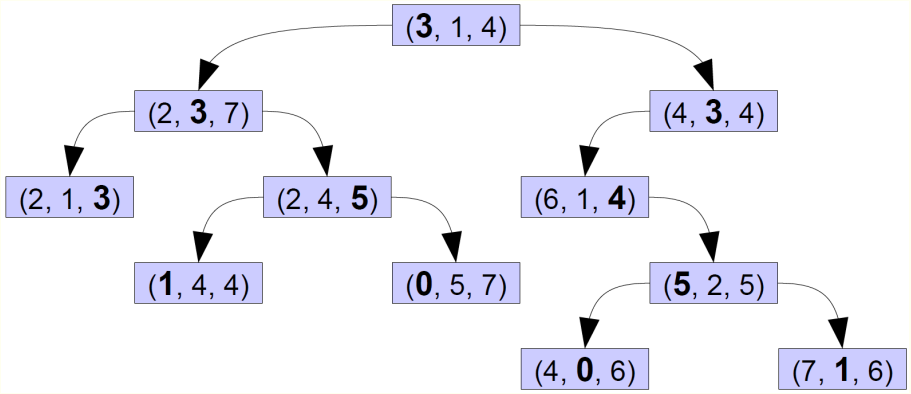
Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ``TDDD86 Lab EL1 redovisning''` och en `git push`. Skicka sedan ett mail till din assistent med ämnet: `[TDDD86] Lab EL1 redovisning .` Se till att filen **KDTree.h** är med.

Kd-träd

I kursen har vi sett många olika containerklasser från STL. En sak samtliga dessa containrar har gemensamt är att de är *exakta*. Ett element är antingen med i ett `set` eller inte. Ett värde uppträder antingen på en specifik plats i en `vector` eller inte. I de flesta tillämpningar är det precis det beteendet vi vill ha. Dock kan man tänka sig situationer då vi inte är intresserade av frågan "finns X i containern?" utan snarare "vilket värde i containern är X mest likt?". Frågor av den typen uppträder ofta inom data mining, maskininlärning och beräkningsgeometri. I den här uppgiften får du implementera en specifik datastruktur kallad för ett *kd-träd* (k-dimensionellt träd) som effektivt understödjer denna operation.

På hög nivå är ett kd-träd en generalisering av ett binärt sökträd till att lagra punkter från ett k-dimensionellt rum. Det går alltså att använda ett kd-träd för att lagra punkter från det kartesiska planet, i tre dimensioner, etc. Du kan också använda ett kd-träd för att lagra biometriskt data genom att , till exempel, representera datat som en ordnad tupel, kanske (längd, vikt, blodtryck, kolesterol). ett kd-träd kan dock inte användas för att samlingar av annat data som till exempel strängar. Notera också att även om ett kd-träd kan lagra data av godtycklig dimension så måste allt data i ett givet kd-träd ha samma dimensionalitet.

Det är lättast att förstå hur ett kd-träd fungerar genom att se ett exempel. Nedan är ett kd-träd som lagrar 3-dimensionellt data:



Notera att på varje nivå i kd-trädet är en särskild komponent av varje nod i fet stil. Om vi noll-indexerar komponenterna är den $(n \bmod 3) : e$ komponenten på nivå n i fet stil. Anledningen till detta är att varje nod agerar som ett binärt sökträd som diskriminerar endast längs den fetstilta komponenten. Till exempel är den första komponenten av varje nod i vänster delträd mindre än den första komponenten i rotnoden, medan första komponenten i varje nod i höger delträd är minst lika stor som rotnodens.

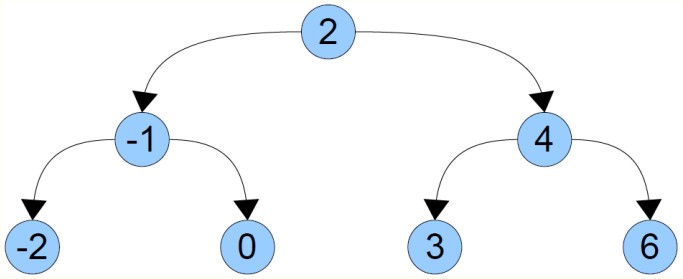
Givet hur kd-trädet lagrar sitt data kan vi effektivt avgöra om en punkt finns i kd-trädet på följande vis. Givet en punkt P, börja i trädets rot. Om rotnoden är P, returnera rotnoden. Om första komponenten i P är strikt mindre än första komponenten i rotnoden, leta efter P i vänster delträd, nu genom att jämföra den andra komponenten av P. Leta annars vidare i höger delträd och jämför den andra komponenten av P. Vi fortsätter med den här processen och ändrar cykliskt vilken komponent som används i varje steg till vi antingen faller ur trädet eller hittar noden i fråga. Insättning i kd-trädet är på samma vis analogt med proceduren i ett binärt sökträd, förutom att varje nivå endast betraktar en del av en punkt.

Den geometriska intuitionen bakom kd-träd

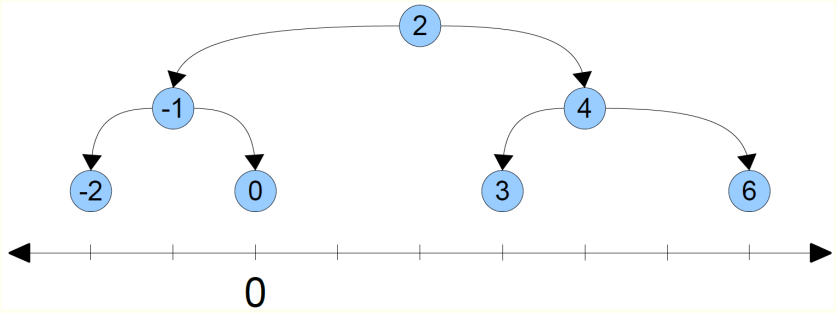
Du kanske undrar varför kd-träd lagrar data på det sätt de gör. Det är trots allt inte uppenbart varför olika koordinater jämförs på olika nivåer i trädet. Det visar sig att det finns en tjusig geometrisk betydelse bakom

denna metod och genom att utnyttja denna struktur är det möjligt att slå upp närmsta grannar på ett extremt effektivt sätt (på tid bättre än $\mathcal{O}(n)$) genom att använda ett kd-träd.

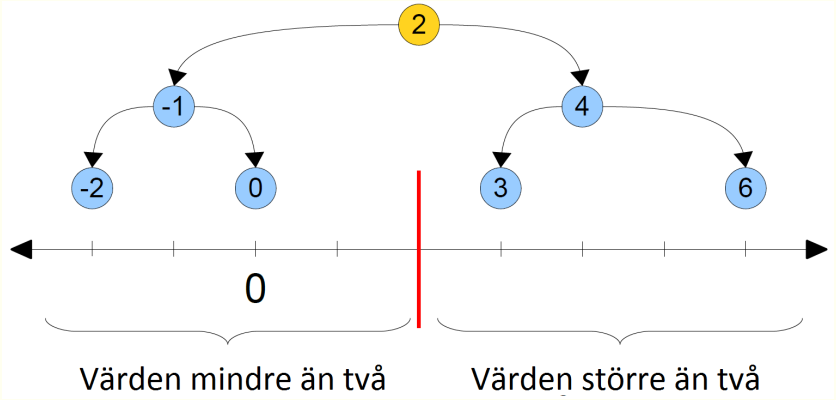
För att göra intuitionen bakom koordinat-för-koordinatjämförelsen klarare återvänder vi till binära sökträd för att utforsla en aspekt av dessa du kanske missat att notera. Betraka ett binärt sökträd där varje nod lagrar ett reellt tal. I denna diskusson använder vi följande träd som referens:



Eftersom det binära sökträdet lagrar en samling reella tal kan vi visa den reella tallinjen tillsammans med det binära sökträdet:

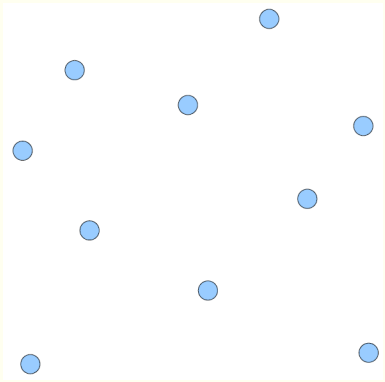


Antag nu att vi traverserar det binära sökträdet och letar efter noll. Vi börjar i rotnoden och kontrollerar om rotnoden innehåller värdet vi letar efter. Eftersom den inte gör det bestämmer vi vilket av de två delträden vi bör fortsätta i och söker rekursivt i det delträdet efter noll. Matematiskt är detta ekvivalent med att dela den reella tallinjen i två regioner — tal mindre än två och tal större än två:

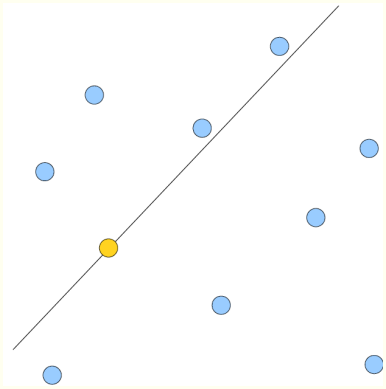


Notera att alla noder i vänster delträd är i den vänstra partitionen och alla noder i höger delträd är i den högra partitionen. Eftersom $0 < 2$ vet vi att om noll finns i trädet måste det finnas i en nod i den vänstra partitionen. Denna insikt är förstås det som gör binära sökträd möjliga. Varje nod definierar någon partition av den reella tallinjen i två segment och vart och ett av nodens delträd är helt innehållet i ett av dessa segment. Sökning i ett binärt sökträd kan alltså tänkas motsvara att kontinuerligt dela rummet i i halvor och sedan fortsätta i halvan som innehåller värdet ifråga.

Det huvudsakliga skälet att nämna ovanstående resonemang är att det går att skala upp till att gälla högredimensionellt data. Antag, till exempel, att vi har följande samling punkter i planet:



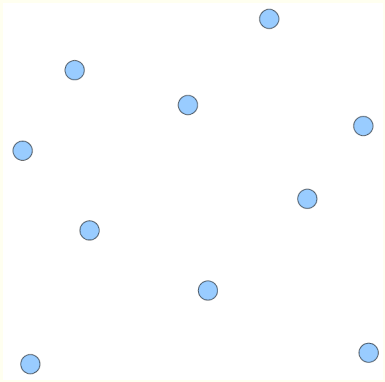
Antag att vi vill bygga ett binärt sökträd av dessa punkter. Om vi använder den bekanta definitionen av binära sökträd skulle vi välja någon av punkterna som rot och sedan bygga ett delträd av kvarvarande noder som är “mindre än” rotnoden och ett delträd av värden “större än” rotnoden. Olyckligtvis finns det inte någon särskilt bra definition av vad det innebär för en punkt i planet att var mindre än en annan. Men låt oss istället betrakta synen på binära sökträd som diskuterades ovan. I ett binärt sökträd delar varje punkt tallinjen i två regioner på ett naturligt sätt. I två dimensioner kan vi dela upp *planet* i två regioner genom att rita en linje genom punkten. Till exempel kan vi dra en linje genom den uppmärkta punkten:



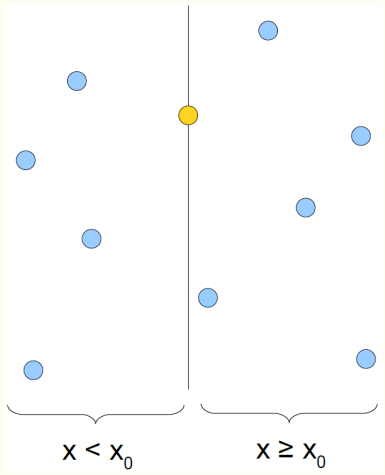
Då har vi delat upp planet i två distinkta regioner. En ovanför linjen och en under linjen. Denna observation ger oss ett sätt att bygga ett binärt sökträd i flera dimensioner. Välj först en godtyckligt punkt och rita en linje (med godtycklig riktning) genom den. Separera sedan kvarvarande punkter i punkter på ena sidan av linjen och av andra sidan av linjen. Konstruera slutligen rekursivt binära sökträd av dessa punkter. Denna teknik är känd som *binary space partitioning* och träd skapade på detta sätt kallas BSP-träd.

Men BSP-träd är inte begränsade till det tvådimensionella planet; samma teknik fungerar för gotyckligt många dimensioner. I tre dimensioner kunde vi partitionera rummet i tvp regioner genom att rita ett plan genom en punkt. När man arbetar med BSP-träd pratar man ofta om termen *delande hyperplan* för att referera till objektet som passerar genom en punkt för att dela rummet i tu. I ett vanligt boinärt sökträd är detta hyperplan bara en vanlig punkt.

Vad har denna diskussion med kd-träd att göra? För att svara på denna fråga måste vi återvända till vår ursprungliga samling punkter i planet:

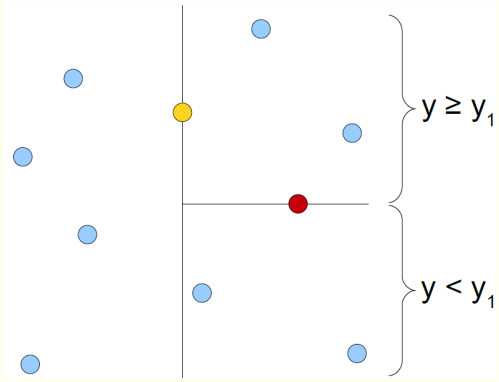


Anag att vi vill bygga ett kd-träd av dessa datapunkter. Vi börjar med att välja någon nod (som vi kan säga finns i (x_0, y_0) för enkelhets skull) och delar datamängden i två grupper, en med punkter vars x-komponenter är mindre än den delande nodens och en vars x-komponenter är större än den delande punktens. Vi kan visualisera delningen på följande vis:

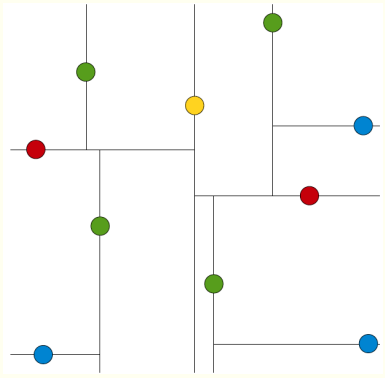


Notera att detta är väsentligen samma sak som att använda ett delande hyperplan genom en av punkterna. I den meningen är ett kd-träd ett specialfall av BSP-träd med en speciell regel för att avgöra vilket delande hyperplan som ska användas. Hur som helst har vi åstadkommit detta utan att behöva skriva kod för att manipulera hyperplan eller halvrymder. All den komplexa geometrin tas om hand implicit.

Låt oss fortsätta bygga det här kd-trädet.Vu bygger rekursivt ett kd-träd i den högra halvrymden (punkterna till höger om centralnoden) genom att välja en punkt och dela datat horisontellt genom den:

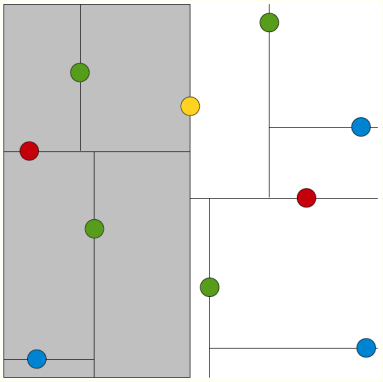


Om vi fullföljer konstruktionen till slutet ser vårt resulterande kd-träd ut så här:

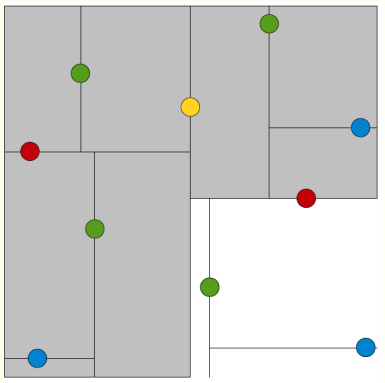


Här är den gula noden rotnod, noder en nivå ned är röda, noder på djup två är gröna och noder på djup tre är blåa.

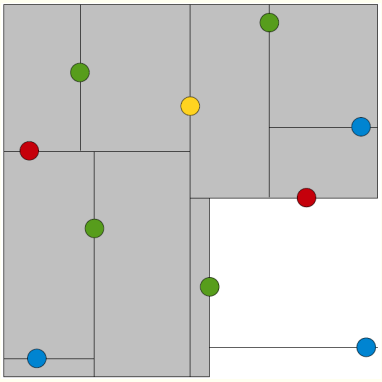
Låt oss undersöka vad som händer när vi slår upp en given punkt i kd-trädet. Detta kommer ge en bättre förståelse för den geometriska intuitionen bakom konstruktionen. Antag att vi letar efter noden längst ned till höger i kd-trädet. Vi startar i kd-trädets rotnod och tar reda på om vår nods x-koordinat är mindre eller större än rotnodens x-koordinat. Detta är ekvivalent med att dela planet vertikalt i rotnoden och sedan fråga vilken halva av planet vår nod finns i. Vår nod råkar finnas i den högra halvan så vi kan därför strunta i alla noder i vänster halva och rekursivt utforska den högra. Detta visas grafiskt nedan, där den gråa regionen motsvarar delar av planet vi aldrig mer tittar på:



Nu kontrollerar vi om vår nod är ovanför eller nedanför den röda noden, vilken är rot i trädet av denna halva. Vår nod finns nedanför den, så vi kastar bort den övre halvan och tittar i den undre halvan:



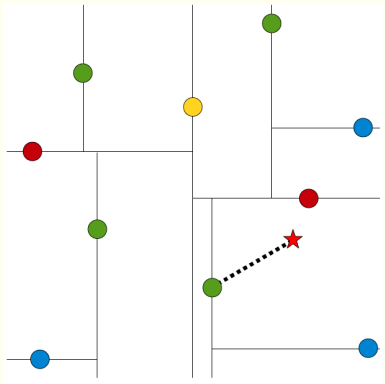
Sedan kontrollerar vi om vi är till höger eller till vänster om den gröna noden som är rot i den här regionen av rummet. Vi finns till höger om den, så vi kan strunta i strimlan till vänster om den gröna noden och fortsätta:



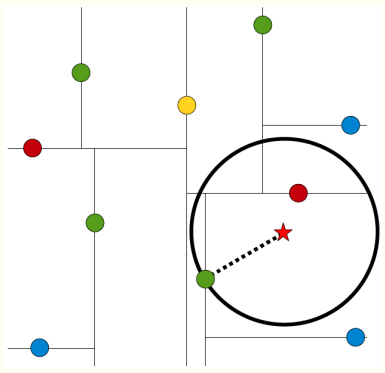
Nu har vi nått noden vi letar efter och sökningen terminerar.

Sökning efter närmsta granne i kd-träd

Utrustade med lite bättre geometrisk intuition för kd-trädet kan vi prata om den mest intressant operationen på kd-träd: uppslagning av närmsta granne. Frågan vi ställer är: givet ett kd-träd och en punkt i rummet (kallad testpunkten), vilken punkt i kd-trädet har kortast avstånd till testpunkten? Innan vi diskuterar den faktiska algoritmen för att göra uppslagningen diskuterar vi intuitionen bakom algoritmen. Antag att vi har en gissning på vilken punkt som är närmst testpunkten. Antag, till exempel, att testpunkten är markerad med en stjärna och att vi tror att dess närmsta granne är punkten sammanbunden med stjärnan med en streckad linje:



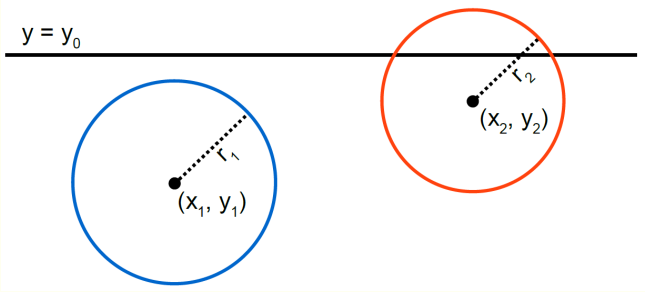
Givet vår gissning av vilken den närmsta grannen är kan vi göra en avgörande observation. Om det finns en punkt i den här datamängden som är närmre testpunkten än vår nuvarande gissning så måste den finnas i cirkeln centrerad i testpunkten som passerar genom den nuvarande gissningen. Denna cirkel visas här:



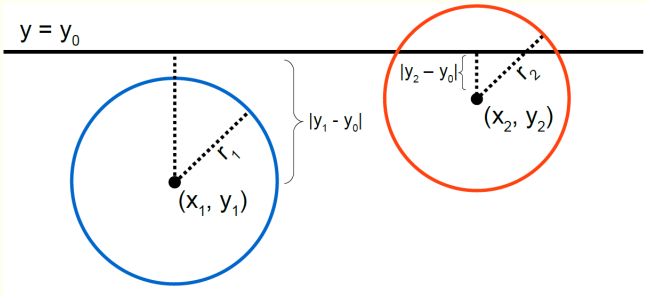
I det här exemplet är denna region en cirkel, men i tre dimensioner skulle den vara en sfär och i allmänhet kallar vi regionen *kandidathypersfären*.

Anledningen till att denna observation är så viktig är att den låter oss trimma vilka delar av trädet som kan innehålla den sanna närmsta grannen. Speciellt kan vi notera att cirkeln finns helt till höger om det delande hyperplanet som går vertikalt genom trädets rotnod. Det betyder att ingen punkt till vänster om trädets rot finns med i kandidathypersfären och kan följdaktligen inte vara bättre än vår nuvarande gissning. När vi väl har en gissning på var den närmsta grannen finns kan vi med andra ord börja utesluta delar av trädet där det faktiska svaret inte kan finnas. Denna generella teknik att utforska en stor sökrymd och trimma bort val baserat på partiella resultat kallas *branch-and-bound*.

Från bilden är det klart att cirkeln av möjliga närmsta grannar inte korsar det mittersta delande hyperplanet men hur kan vi avgöra detta matematiskt? Givet en cirkel och en linje (eller mer generellt, en hypersfär och ett hyperplan) är det, i allmänhet, lite knepigt att avgöra om cirkeln skär linjen. Som tur är förenklar faktumet att vi valt att alla delande hyperplan ska vara i linje med koordinataxlarna saken betydligt. Nedan är en godtycklig linje och två cirklar, en som skär linjen och en som inte skär linjen:



Betrakta nu avståndet från respektive cirkels mittpunkt till linjen $y = y_0$. Detta är helt enkelt absolutbeloppet av skillnaden mellan cirkelns y-koordinat och y_0:



Lägg märke till att avståndet $|y_1 - y_0|$ från mittpunkten av den blåa cirkeln är större än cirkelns radie, så cirkeln skär inte linjen. Å andra sidan är avståndet från mittpunkten av den röda cirkeln till linjen mindre än den cirkelns radie, så någon del av cirkeln skär faktiskt linjen. Detta ger oss ett generellt villkor för att avgöra om kandidathypersfären skär ett särskilt delande hyperplan. Speciellt, givet ett kd-träd med punkterna $(a_0, a_1, a_2, \ldots, a_k)$ och en hypersfär med radie r centrerad i $(b_0, b_1, b_2, \ldots, b_k)$, om noden partitionerar punkter baserat på deras i:te komponent så skär hypersfären nodens delande hyperplan endast om $|b_i - a_i| < r$.

För att sammanfatta:

- Givet en gissning på vilken nod som är den närmsta grannen kan vi konstruera en kandidathypersfär centrerad i testpunkten som går genom gissningspunkten. Den närmsta grannen till testpunkten måste finnas i denna hypersfär.
- Om denna hypersfär finns helt på en sida av ett delande hyperplan kan ingen av punkterna på den andra sidan om hyperplanet finnas i hypersfären och kan alltså heller inte vara den närmsta grannen.
- För att avgöra om kandidathypersfären skär ett delande hyperplan som jämför koordinat i räcker det att kontrollera om $|b_i - a_i| < r$.

Tagna tillsammans ger dessa observationer oss följande algoritm för att hitta den närmsta grannen till en testpunkt:

```
Let the test point be (a0, a1, ..., ak).

Maintain a global best estimate of the nearest neighbor, called 'guess.'
Maintain a global value of the distance to that neighbor, called 'bestDist'

Set 'guess' to NULL.
Set 'bestDist' to infinity.

Starting at the root, execute the following procedure:
   if curr == NULL
      return

   /* If the current location is better than the best known location,
    * update the best known location.
    */
   if distance(curr, guess) < bestDist
      bestDist = distance(curr, guess)
      guess = curr

   /* Recursively search the half of the tree that contains the test point. */
   if ai < curri
      recursively search the left subtree on the next axis
   else
      recursively search the right subtree on the next axis

   /* If the candidate hypersphere crosses this splitting plane, look on the
    * other side of the plane by examining the other subtree.
    */
   if |curri - ai| < bestDist
   recursively search the other subtree on the next axis
```

Intuitivt fungerar ovanstående procedur genom att gå ned till ett löv i kd-trädet som om vi sökte efter testpunkten. När vi börjar nysta upp rekursionen oh gå upp längs trädet kontrollerar vi om varje nod är bättre än vår uppskattning. Skulle det vara fallet uppdaterar vi vår uppskattning till den nuvarande noden. Slutligen kontrollerar vi om kandidathypersfären baserat på vår nuvarande gissning kan skära den nuvarande nodens delande hyperplan. Om den inte gör det kan vi utesluta alla punkter på andra sidan om det delande hyperplanet och gå upp till nästa nod i trädet. I annat fall måste vi titta på andra sidan av trädet för att se om det finns närmre punkter där.

Denna algoritm kan visas ha exekveringstid $\mathcal{O}(\log n)$ för ett balanserat kd-träd med n punkter givet att dessa punkter är jämnt utspridda. I det värsta fallet måste dock hela trädet genomsökas. Med ett lägre antal dimensioner som i det kartesiska planet eller det tredimensionella rummet är dock det värsta fallet mycket ovanligt.

k närmsta grannarna och begränsade prioritetsköer

I föregående diskussion har vi endast behandlat problemet att hitta den närmsta grannen till en testpunkt. En intressantare fråga är att, givet en testpunkt och något heltal k, hitta de k närmsta grannarna till denna punkt. Denna typ av sökning förkortas ofta som k-NNsökning. det visar sig att algoritmen ovan lätt kan anpassas till k-NNsökning i stället för 1-NNsökning. Algoritmen är nästan identisk förutom att i stället för att hålla reda på enbart den bästa punkten håller vi reda på de k bästa punkterna vi sett hittills.

Innan vi beskriver algoritmen introducerar vi en speciell datastruktur, en *begränsad prioritetskö* eller *BPQ*. En begränsad prioritetskö är lik en vanlig priosítetskö förutom att det finns en fix övre gärna på antalet element som kan lagras i prioritetskön. Om prioritetskön är full när ett element läggs till i kön kastas elementet med högst prioritetsvärde ut från kön. Antag, till exempel, att vi har en BPQ med kapacitet fem:

Värde	A	B	C	D	E
Prioritet	0.1	0.25	1.33	3.2	4.6

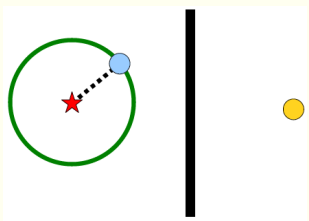
Antag att vi vill sätta in elementet F med prioritet 0.4 i denna BPQ. Eftersom denna begränsade prioritetskö har kapacitet fem sätts elementet in, men elementet med lägst prioritet (E) kastas då ut:

Värde	A	B	F	C	D
Prioritet	0.1	0.25	0.4	1.33	3.2

Antag nu att vi vill sätta in G med prioritet 4.0 i denna BPQ. Eftersom G:s prioritetsvärde är högre än det största prioritetsvärdet i BPQ:n kommer G omedelbart att kastas ut vid insättning. Givet tillgång till en BPQ kan vi genomföra k-NNsökning i ett kd-träd på följande vis:

```
Let the test point be P = (y0, y1, ..., yk).

Maintain a BPQ of the candidate nearest neighbors, called 'bpq'
Set the maximum size of 'bpq' to k

Starting at the root, execute the following procedure:
   if curr == NULL
      return

   /* Add the current point to the BPQ. Note that this is a no-op if the
    * point is not as good as the points we've seen so far.
    */
   enqueue curr into bpq with priority distance(curr, P)

   /* Recursively search the half of the tree that contains the test point. */
   if yi < curri
      recursively search the left subtree on the next axis
   else
      recursively search the right subtree on the next axis

   /* If the candidate hypersphere crosses this splitting plane, look on the
    * other side of the plane by examining the other subtree.
    */
   if:
      bpq isn't full
         -or-
      |curri - yi| is less than the priority of the max-priority elem of bpq
   then
      recursively search the other subtree on the next axis
```

Det är två ändringar som skiljer den här algoritmen från den för 1-NNsökning. För det första, när vi bestämmer om vi ska titta på andra sidan om det delande planet så använder vi som radie för kandidathypersfären avståndet mellan testpunkten och punkten med högst prioritetsvärde i BPQ:n. Anledningen till detta är att när vi säker efter de k närmsta grannarna så behöver kandidathypersfären innefatta samtliga k av dessa grannar, inte enbart den närmsta. Den andra större ändringen är att när vi överväger om vi ska titta på andra sidan av det delande planet tar vårt belut hänsyn till huruvida BPQ:n redan innehåller minst k punkter. Detta är extremt viktigt! Om vi trimmar bort delar av trädet innan vi gjort k gissningar kan vi råka kasta bort en av de närmsta punkterna. Betrakta följande situation:



Antag nun att vi vill göra 2-NNsökning med testpunkten markerad med en stjärna. Vi undersöker rekursivt delträdet till vänster om det delande planet och hittar den blåa punkten som kandidat till närmsta granne. Eftersom vi inte hittat två närmsta grannar ännu måste vi titta på andra sidan om det delande planet, trots att kandidathypersfären inte skär det delande planet.

Uppgiften

Din uppgift är att implementera en klass som representerar ett kd-träd, kallad `KDTree`. Klassen låter en klient bygga upp ett kd-träd, söka efter medlemmar i trädet och exekvera k-NNsökningar. När du utför uppgiften får du tillfälle att öva på klassdesign, `const`-korrekthet, mallar, kopiering, operatoröverlagring och undantagshantering. Mängden kod du behöver skriva är inte särskilt stor — i storleksordningen tvåhundra rader — men den kräver att du har ett bra grepp om möjligheterna C++ ger oss.

Nedan följer en lämplig arbetsgång:

Steg 0

Konfigurera projektet och bekanta dig med den givna koden.

Steg 1: Implementera basfunktionalitet

Implementationen av `KDTree` du kommer att skriva är en liten variant av det som beskrivits ovan, där vi associerar extra data med varje punkt. På ett sätt kommer ditt `KDTree` att vara en lite tjusigare `map` från punkter i rummet till värden.

Nedan finns en partiell specifikation av klassen `KDTree` med de funktioner du behöver skriva för att få igång basfunktionaliteten.

```
template <size_t N, typename ElemType> class KDTree {
public:
   KDTree();
   ~KDTree();

   size_t dimension() const;
   size_t size() const;
   bool empty() const;

   void insert(const Point<N>& pt, const ElemType& value);

   bool contains(const Point<N>& pt) const;
   ElemType& operator[] (const Point<N>& pt);
   ElemType& at(const Point<N>& pt);
   const ElemType& at(const Point<N>& pt) const;
};
```

Du kanske har lagt märke till att `KDTree` har en ovanlig mallsignatur:

```
   template <size_t N, typename ElemType> class KDTree
```

Implementationen av `KDTree` är parametriserad över en `size_t` och en typ. Heltal som templateparametrar beter sig precis som vanliga typparametrar. Om du vill skapa ett `KDTree` som avbildar punkter i tre dimensioner på strängar kan du deklarera den som

```
   KDTree<3, string> myKDTree
```

Nycklarna i kd-trädet är objekt av typen `Point<N>`, där `N` är kd-trädets dimension. Bland startfilerna finns en färdig implementation av `Point`; den beter sig som en STL `vector<double>` av fix längd. Till exempel:

```
Point<3> pt;
pt[0] = 137.0;
pt[1] = 42.0;
pt[2] = 2.71828;
```

Titta gärna på implementationen i `Point.h` för att se vilken funktionalitet som erbjuds.

Börja nu med att implementera följande medlemsfunktioner i `KDTree`:

`KDTree();`	Constructs a new, empty `KDTree`.
`~KDTree();`	Destroys the `KDTree` and deallocates all its resources.
`size_t dimension() const;`	Returns the dimension of the points stored in the `KDTree`. (This is the value of the template parameter `N`).
`size_t size() const;` `bool empty() const;`	Returns the number of elements stored in the `KDTree` and whether or not it is empty, respectively.

`void`
`insert (const Point<N>& pt,`
`const ElemType& value);`
	Inserts the specified point into the `KDTree` with associated value `value`. If the point already exists in the `KDTree`, the old value is overwritten.

`bool` `contains(const Point<N>& pt) const;`	Returns whether the specified `Point` is contained in the `KDTree`.
`ElemType&` `operator[] (const Point<N>& pt);`	Returns a reference to the value associated with the point `pt`. If the point does not exist in the `KDTree`, it is added with the default value of `ElemType` as its value, and a reference to this new value is returned. This is the same behavior as the STL `map`'s `operator[]`. Note that this function does not have a `const` overload because the function may mutate the tree.

`ElemType&`
`at(const Point<N>& pt);`
 	
`const ElemType&`
`at(const Point<N>& pt) const;`
	Returns a reference to the value associated with the point `pt`, if it exists. If the point is not in the tree, then this function throws an `out_of_range` exception.
	
	This function *is* `const`-overloaded, since it does not change the tree.

Notera att de fyra sista funktionerna alla gör någon sorts sökning i kd-trädet efter ett specifikt värde, med enda skillnad i beteende när punkten inte finns i trädet. `contains` returnerar `false`, `operator[]` lägger till ett nytt element och `at` kastar ett `out_of_range`-undantag. I stället för att skriva koden för att traversera trädet fyra gånger och specialisera beteendet när ett element inte hittas rekommenderar vi starkt att skriva en hjälpfunktion som söker i trädet efter en specifik punkt och returnerar en pekare till noden som innehåller den. Här är, till exempel, en enkel implementation av `contains` som förutsätter existensen av en hjälpfunktion `findNode`:

```
template <size_t N, typename ElemType>
bool KDTree<N, ElemType>::contains(const Point<N>& pt) const {
    return findNode(pt) != nullptr;
}
```

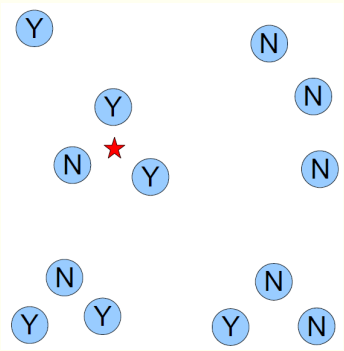
För att testa att din kod fungerar kan du köra den första uppsättningen tester i projektets `test-harness`. Det kan också vara lämpligt att lägga till dina egna tester.

Steg 2: Implementera uppslagning av närmsta granne

Nu när du har basfunktionaliteten på plats är det dags att implementera k-NNsökning. Din nästa uppgift är att implementera funktionen `kNNValue`, vilken har följande signatur:

```
    ElemType kNNValue(const Point<N>& pt, size_t k) const;
```

Den här funktionen tar in en punkt i rummet och ett antal eftersöka närmsta grannar. Den ska genomföra en k-NNsökning i kd-trädet med `pt` som testpunkt. Efter att den gjort detta har funktionen en samling av de k närmsta punkterna i rummet tillsammans med `ElemType`-värdena associerade med dem. Returvärdet för funktionen ska vara det mest frekvent förekommande värdet associerat med de k närmsta grannarna till testpunkten. Om fler värden är mest vanligt förekommande går det bra att returnera vilket som helst av dem. Betrakta följande mängd av punkter tillsammans med den indikerade testpunkten:



Om vi gjorde en 3-NNsökning skulle `kNNValue`-funktionen returnera "Y".

Algoritmen för att göra k-NNsökning förutsätter existensen av en begränsad prioritetskö. Bland startfilerna finns en klass `BoundedPQueue` som implementerar just detta.

Testramverket innehåller två funktioner som testar denna funktion. Se till att din funktion fungerar innan du går vidare till nästa del.

Steg 3: Implementera djup kopiering

Som den här skriven har `KDTree` en konstruktor, men ingen kopieringskonstruktor eller tilldelningsoperator. För att undvika krascher behöver du nu implementera en kopieringskonstruktor och en tilldelningsoperator för `KDTree`-klassen:

```
KDTree(const KDTree& other);
KDTree& operator= (const KDTree& other);
```

Testramverket innehåller två test som stressar kopieringsfunktionerna. Säkerställ att din implementation klarar dessa tester.