Augmented Data Structures

Jad Hamdan and Tyler Kastner

March 2, 2020

This is the augmented transcript of two lectures given by Luc Devroye in his Data Structures and Algorithms classes (COMP 251/252) at McGill University.

Introduction

In certain situations, our standard data structures (such as a linked list, or binary search tree), are enough for our needs, but this is not always the case. Instead of creating an entirely new data structure however, what we can do is 'augment' an existing data structure, by storing additional information in it. Some examples of augment data structures are:

- Data participating in multiple data structures
- Search trees & lists
- Order statistics trees
- Interval trees

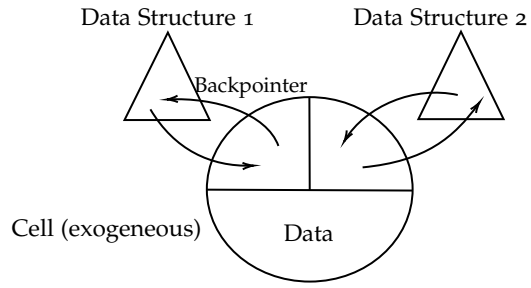


Figure 1: Visualization of data participating in multiple data structures

Combining Red-Black Tree and Linked List

Within a red-black tree, it may be interesting to store a 'next' and a 'previous' pointer with each node, so we can easily browse it. The next pointer points to the next node in the ordering of the keys. We will see that if we keep track of these pointers from the beginning of our tree, it is not difficult to accomplish this.

As seen in figures 3 and 4, it is very straightforward to update the pointers while doing our ordinary operations, they in fact take constant time. Therefore we will be able to do all our normal red-black tree operations, without adding extra time to these operations. This is what we will aim to do when augmenting data structures. Note that the data live in two data structures, a red-black tree and a linked list.

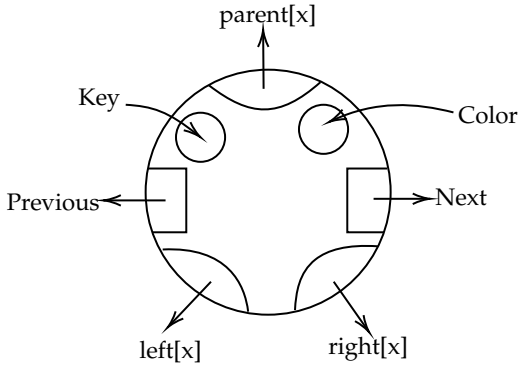
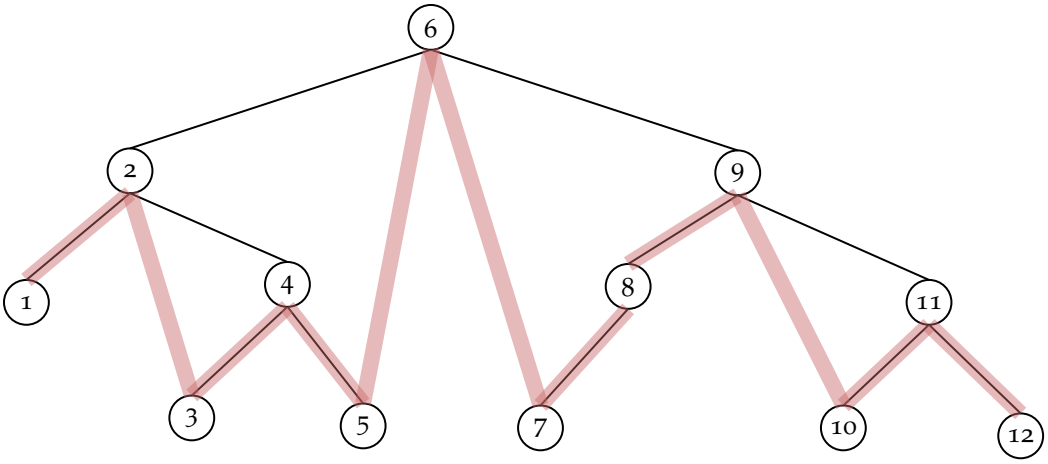


Figure 2: The cell of a combined red-black tree and linked list



Order Statistics ADT

The ADT "Order Statistics" supports the standard dictionary operations (INSERT, DELETE, SEARCH), and two new operations:

- SELECT(k, D): retrieve the kth smallest element of D
- RANK(x,D): return the rank of the item pointed to by x in D.

We can imagine SELECT and RANK are opposite operations. In the former, we are inputting a rank and outputting a node, while in the latter we are inputting a node and outputting a rank. We will implement this data structure via an Order Statistics Tree, which is an augmented red-black tree. We will create the red-black tree with cells as seen in Figure 2, where "size" is the number of nodes in the subtree.

The new important field to keep track of is *size*. We have two important tasks: determining whether we can still perform INSERT, DELETE in $\mathcal{O}(\log n)$, as well as perform SELECT and RANK in $\mathcal{O}(\log n)$.

To maintain *size* during INSERT and DELETE, all we need to do is change the *size* associated with all the ancestors of the inserted or deleted node. If we perform INSERT, we add one to all ancestors, and if we DELETE, we subtract one from all ancestors (as seen in figure 6). When we rotate, the only node whose *size* field changes is the one we are rotating, as can be seen in figure 7. Therefore, this only adds $\mathcal{O}(\log n)$ time to these operations, and they all still run in $\mathcal{O}(\log n)$.

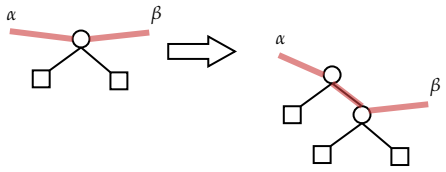


Figure 3: Upkeep of list during INSERT.

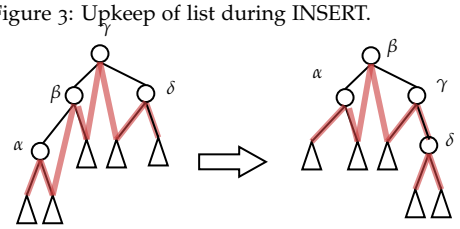


Figure 4: Upkeep of list during ROTATE

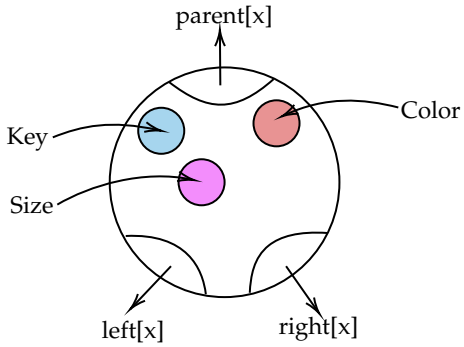


Figure 5: The cell of an order statistics tree.

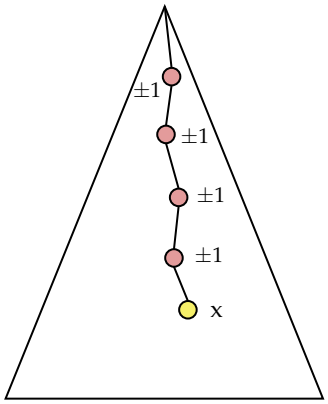


Figure 6: The strategy for inserting and deleting.

Note: by an extension of the above arguments, it is straightforward to maintain fields of the form $f(u) = \sum_{u \in subtree} f(key[u])$, or $f(u) = \max_{u \in subtree} f(key[u])$ (respectively *min*).

SELECT(k, t)

```
1  // t is a pointer to the root of the tree
2  if left[t] = nil
3        r = 1 // r is the rank of the root
4  else r = 1 + size[left[t]]
5  case
6        r = k: return t
7        k < r: return SELECT(k, left[t])
8        k > r return SELECT(k-r, right[t])
```

This runs in $\mathcal{O}(\log n)$ time, as desired.

RANK(x, t)

```
1  // x is a pointer to a node, t is a pointer to root
2  if left[x] = nil
3        r = 1
4  else r = 1 + size[left[x]]
5  y = x // Travelling pointer
6  while y ≠ t
7        if right[parent[y]] = y
8              r = r + 1 + size[left[parent[y]]]
9              // rank = 1 + number of green items
10             y = parent[y]
11 return r
```

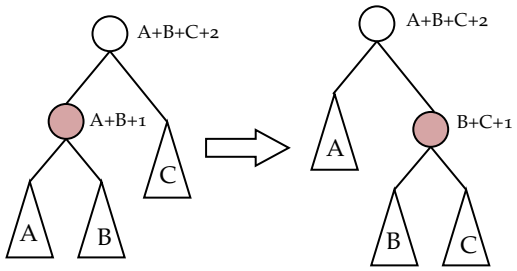


Figure 7: Subtree sizes are shown next to each node involved in a rotation.

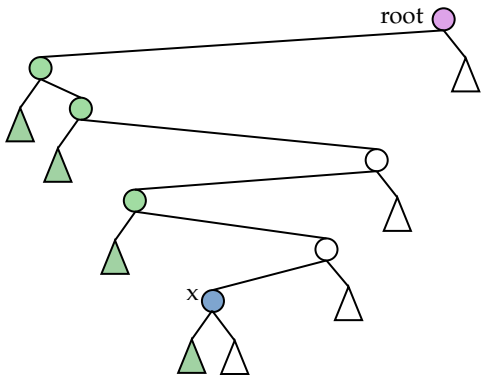


Figure 8: The idea behind the algorithm for RANK. The rank of x is the number of elements to the left of it, so we will count all of the green nodes in the figure.

Interval Trees

Used to store intervals, Interval Trees make the task of finding overlap between said intervals particularly efficient. We will begin by introducing a problem that will serve as motivation for the technical details to come:

Placing rectangles on a surface with no overlap
We are given a finite rectangular surface and a placement of n smaller rectangles on this surface (more specifically, a list of a_i, b_i, x_i and y_i for each rectangle R_i, as defined in the Figure 9). How can we check that no two rectangles overlap? *We could encounter such a problem while designing a chip, for instance.*

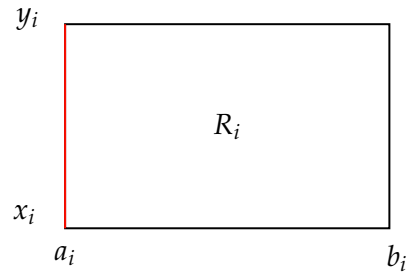


Figure 9: Visual explanation of what a_i, b_i, c_i, and d_i correspond to for a rectangle R_i. R_i's left side (red) is the interval we're interested in.

In a naïve approach, one could just compare all pairs of rectangles to check for overlap. This would cost $\Theta(n^2)$. time. We will outline. a faster method that runs in $\mathcal{O}(n \log n)$ time.

We introduce the *sweepline* problem solving paradigm, which consists in replacing one of the problem's dimensions with time, effectively reducing its dimensionality by one. We do so with our surface's x-axis; one can imagine a *sweepline* traversing the surface from left to right and encountering rectangles along the way. We refer to these encounters as *births*, as opposed to *deaths* which occur when a rectangle is no longer touched by the *sweepline*.

Assume, for now, that we have an arbitrary data structure capable of storing intervals. Using the latter, along with the *sweepline* paradigm, the following algorithm can check if a placement is valid. Start from the left, and "*sweep*" through the surface. When faced with a *birth*, insert (in our data structure) the interval formed by y-coordinates of the rectangle in question. In case of a *death*, delete the corresponding interval from our data structure. At every birth, check if the newly added interval causes any overlap and terminate if it is the case (by doing so, we discretize time and limit ourselves to one check per rectangle).

SWEEPLINE(a, b, x, y) *//a is the sequence of all a_i's, b of b_i's, etc.*

```
1  MAKENULL(t) //create empty interval tree
2  L ← SORT(a,b) //returns a sorted list containing all a_i's and b_i's
3  for i = 1 to 2n do
4  // to simplify, we assume that we have a way of accessing x_i and y_i
5      if L[i] is a birth
6          if OVERLAP((x_i,y_i),t) //this checks if [x_i,y_i] intersects any existing interval in t
7              exit "INVALID PLACEMENT"
8          else INSERT((x_i,y_i),t)
9      if L[i] is a death
10         DELETE((x_i,y_i),t)
11 exit "VALID PLACEMENT"
```

A brief explanation of how "sweeping" translates into code: we sort the a_i's and b_i's to get an increasing sequence that effectively represents the x-axis (more specifically, the points at which births/deaths occur), and iterate through the sequence to "traverse".

We will see below how to implement the interval tree such that INSERT, DELETE and OVERLAP each take $\mathcal{O}(\log n)$ time. We established that there are at most n OVERLAP checks (where n is the number of rectangles to be placed). Each rectangle is inserted and deleted once from the Interval Tree; this adds up to $\mathcal{O}(n \log n)$ time.

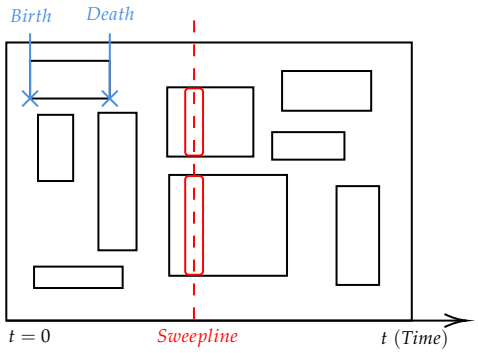


Figure 10: Visualization of the sweepline (red) traversing the surface, along with examples of points where a birth/death occurs (blue).

The initial sorting step yields another $\mathcal{O}(n \log n)$ term, giving a total run-time $\mathcal{O}(n \log n)$, as desired.

The description above reveals the following abstract data type requirements:

- Objects: Intervals
- Operations: INSERT, DELETE, OVERLAP

OVERLAP checks if a given interval intersects any interval already stored in the data structure.

Implementation:
Interval trees are augmented Red-Black trees. As displayed in Figure 11, the nodes are start off as RB-tree nodes to which we add three attributes. The first two are *Low* and *High*, referring to the beginning and end point of the interval stored in the node. We will use *Low* as a key for the node. Next, we have the *Max* attribute, referring to the largest key in this node's subtree. Using this information, the OVERLAP algorithm is as follows:

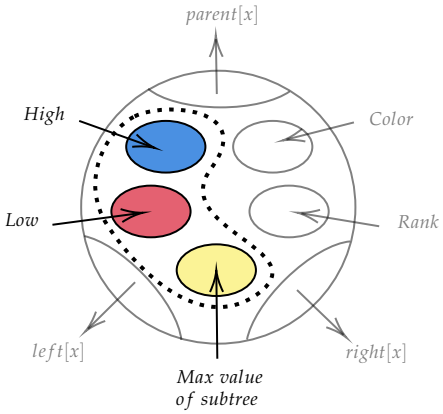


Figure 11: Example of an interval tree node (added attributes in color).

OVERLAP(a, b, t) *//checks if [a,b] overlaps any interval stored in the interval tree with root t.*

```
1  if  t = nil
2      return FALSE
3  else if  [a, b] ∩ [low[t], high[t]] ≠ ∅
4      return TRUE
5  else if left[t] ≠ nil and a ≤ Max[left[t]]
6      return OVERLAP(a, b, left[t])
7  else
8      return OVERLAP(a, b, right[t])
```

The reasoning behind this is simple: if the current node t is *nil*, then there is nothing to be done and we return "false". We then check if the new interval (provided as input) intersects with the interval stored in t, in which case we return "TRUE". If $Low[t]$ is smaller than the largest "high" field in its left sub-tree, we recurse on the left. Otherwise, the interval in t cannot possibly intersect with any interval in its left sub-tree, allowing us to recurse on the right (completely ignoring the left).

The number of recursive calls thus at most matches the height of the tree, which is $\mathcal{O}(\log n)$ since we have a Red-Black tree. OVERLAP therefore runs in $\mathcal{O}(\log n)$ time.

Recall the previously mentioned *sweepline* algorithm:

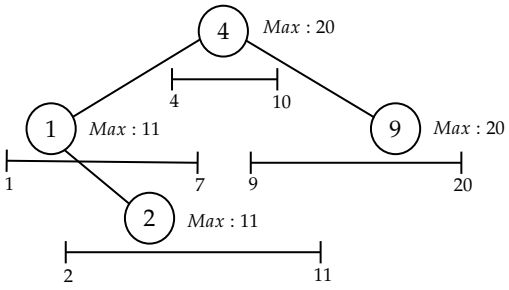


Figure 12: Example of an Interval Tree. Each node contains an interval (depicted below) and said interval's lowest point is used as the key.

Level Linking in Red-Black Trees

We would like to implement fast browsing. Some operations which we would like to be able to perform are:

- INSERT, DELETE
- SEARCH
- NEXT
- k-NEXT (skip over k items)
- PREVIOUS
- k-PREVIOUS (same idea as k-NEXT)

To achieve this, we will use a data structure known as a level-linked red-black tree. Recall the 2-3-4 tree view of a red-black tree, where black nodes live in different 'levels'. We will augment the red black tree by adding a linked list at each level of black nodes, as seen in figure 2. This data structure will have red cells unchanged, and black cells the same as the augmented cells of the combined red-black tree and linked list (figure 2).

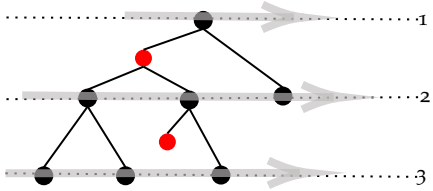


Figure 13: The 2-3-4 view of a level-linked red black tree

The main idea we need to implement is given a key k, we must efficiently find a way to search for y such that key$[y] = k$. We will assume we start at a node x, and key$[x] < k$. We would like to take advantage of our data structure, and so beginning at node x, we will advance up the tree (towards root) until we find a point whose right neighbour "overshoots y", and call this point z (z is the first point such that $[\text{next}[z]] > k$). We will call this neighbour z^*, and from here we will traverse down the tree and search for y (standard search). We might have a problem however, if x were the root and y was the end of a left-going 1-ary tree beginning at right$[x]$. This would be undesirable, as it would take $\mathcal{O}(\log n)$ to search for y, even though $|\text{Rank}(y) - \text{Rank}(x)| = 1$.

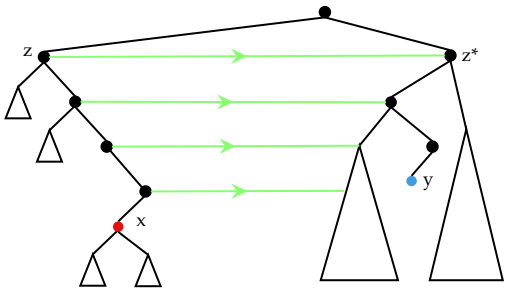


Figure 14: The idea behind searching for key k

As a remedy, we will slightly alter our algorithm as follows:

(1) $x = \text{next}[x]$

(2) from x, continue search as before.

Exercise 1. Prove that Time $\leq 1 + \mathcal{O}(\log(\text{Rank}(y) - \text{Rank}(x)))$

k-d Trees

k-d Trees are binary search trees that are used to partition $\mathbb{R}^k$. They store k-dimensional data. For simplicity and the ability to visualize,

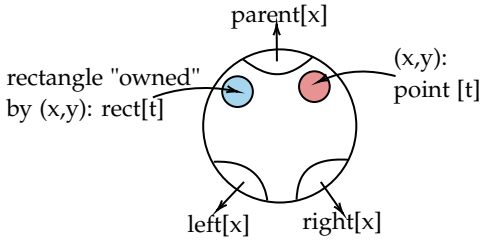
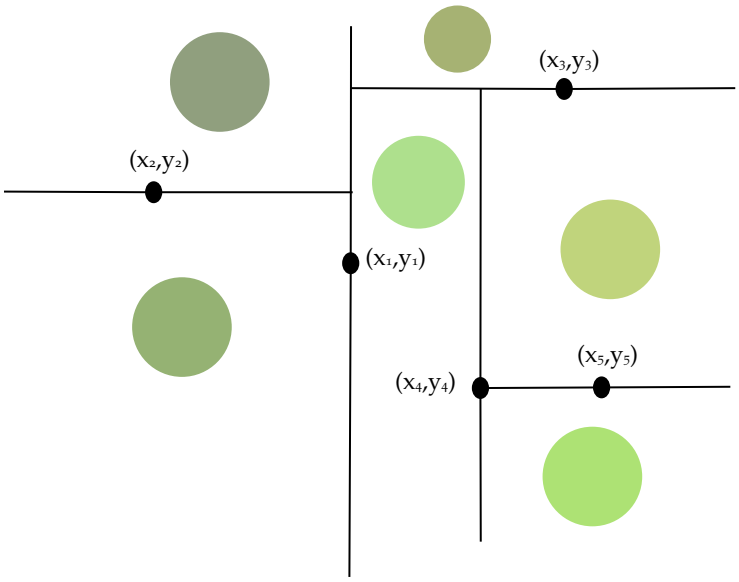
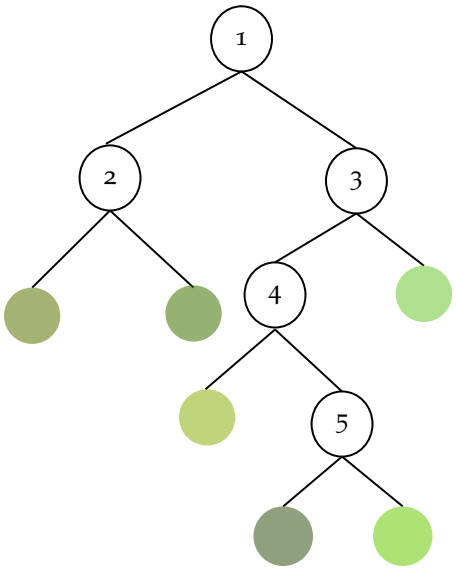


Figure 15: The cell of a k-d tree.

we will consider $k = 2$ (for $k = 1$, we obtain a standard binary search tree). Denote the input by $(x_1, y_1), ..., (x_n, y_n)$.

The first data point, (x_1, y_1) partitions the space with respect to x_1. Recursively on each side, we choose a point, and partition the space with respect to the y-entry this time. Each time we recurse, we cycle through which dimension we use as our key to split. To build the tree, each cut corresponds to an internal node of the tree, and every leaf in the k-d tree corresponds to a region in our final space.





An example of a 2-d tree. Cuts passing through the points (x_k, y_k) are represented by the internal nodes labelled k, and the final regions in the space are represented by leaves of the same colour.

Operations on 2-d trees:

- INSERT, DELETE
- SEARCH for (x, y)
- SEARCH for $(x, *)$
- SEARCH for $(*, y)$
- RANGE SEARCH(R, t) [1]

To accomplish Range Search, we recursively visit all subtrees whose region intersects R, and output all points (x_i, y_i) which lie in R.

Each node x stores left and right pointers, a data entry, point$[x]$, a cut direction, dir$[x]$ and a rectangle Rect$[x]$, i.e., the rectangle that is cut by point$[x]$.

[1] Given a rectangle $R = [a, b] \times [c, d]$ return all points inside this rectangle

Outline of Range Search Algorithm :

Given: rectangle $R = [a, b] \times [c, d]$, tree t.

```
RANGE SEARCH(R, t)
1  if t ≠ nil and R ∩ Rect[t] ≠ nil
2       if point[t] ∈ R
3            Output point[t]
4       RANGE SEARCH(R, left[t])
5       RANGE SEARCH(R, right[t])
```

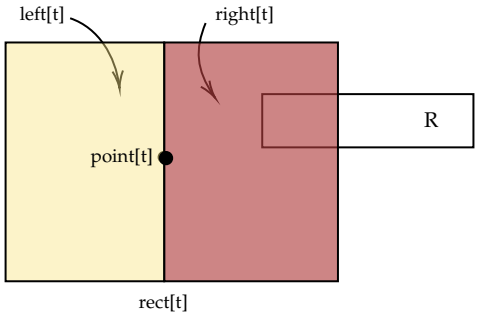
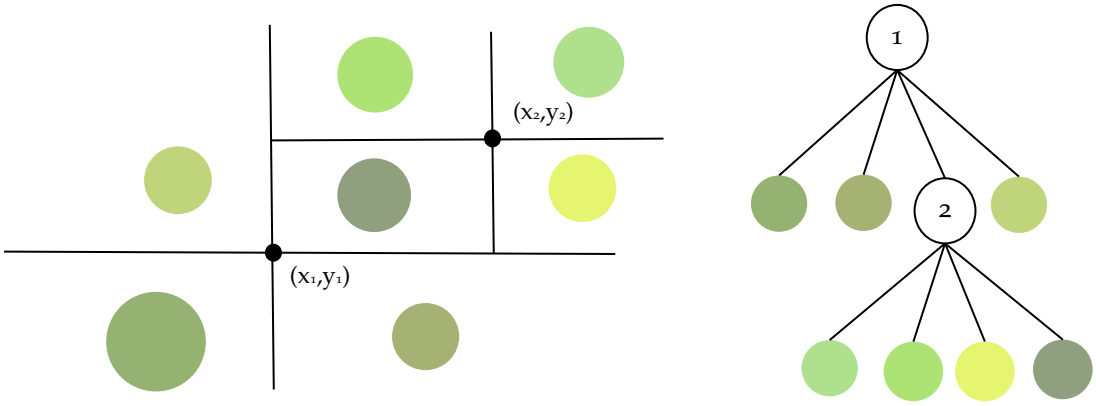


Figure 16: A visualization of Range Search in 2-d

Quadtrees

Quadtrees are 2^d-ary trees similar to k-d trees. The main difference between these and k-d trees is that instead of partitioning using lines, we use quadrants. We can perform Range Search on these as well. Quadtrees are popular in computer graphics.



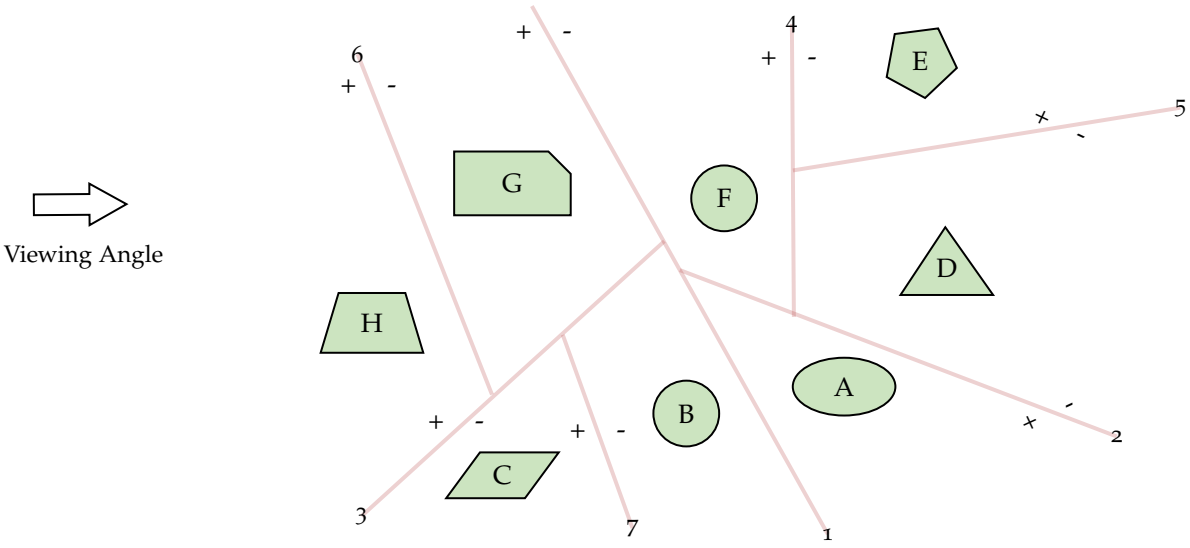
The construction of a quadtree from two points in $\mathbb{R}^2$.

BSP Trees

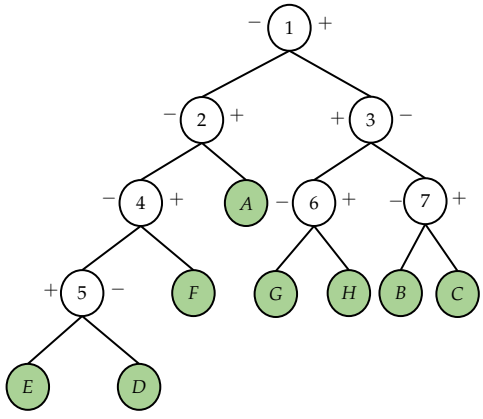
Binary Space Partition trees, also known as BSP trees, are binary trees used in computer graphics. They are used to represent objects living in space using a tree. We will see that the construction of this tree allows us to easily render a scene of this object from a given viewpoint. First, what is the need for such a tree? Imagine you are taking using a camera to take a picture of a scene, of different objects in space. These objects will appear to be projected onto the lens, and the objects at a deeper depth might be hidden by the closer objects. The idea of BSP trees is to address the hidden object problem.

The BSP tree is an augmented binary tree. We do not need to make it a search tree since whether or not it is balanced is not a key issue.

To partition the space, we begin with a "camera angle" which points towards the objects. We will use this to sort. Begin by drawing a hyperplane that separates the objects. We do two things to this hyperplane: first we mark each side with a "+" and a "-" (the "+" side represents which side is facing the front of the camera), and we also mark it with a number to represent which hyperplane cut it is. We then recurse into each region cut into by the hyperplane, and we stop when each object is in its own region. This process is visualized in the figure below.



To build our tree from this, we treat each hyperplane as a node, and after placing a node, we recurse into each half of the space cut by the hyperplane, and repeat the process. When there are no more hyperplanes and only objects, we place the object as the node, so that the leaves are objects. For the above figure, the tree built from it is below.



To paint this as a scene, first paint the objects furthest back. This is the idea behind Painter's Algorithm, which outputs a listing of the objects in the order in which they should be painted. Simply perform a traversal in "- +" order. That is, given a non-leaf node x, visit its "-" subtree before its "+" subtree. When x is a leaf, x represents an object, which should be projected to the viewing plane and rendered. For the above tree, this traversal would give us "D-E-F-A-B-C-G-H", which coincides with the depths of the objects in the figure.

References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.