CSCI-1200 Data Structures — Spring 2022
Lecture 26 – Hybrid / Variant Data Structures

26.1 The Basic Data Structures

This term we've covered a number of core data structures. These structures have fundamentally different memory layouts. These data structures are classic, and are not unique to C++.

- array / vector
- linked list
- binary search tree
- hash table
- binary heap / priority queue

26.2 A Few Variants of the Basic Data Structures

Many *variants* and *advanced extensions* and *hybrid* versions of these data structures are possible. Different applications with different requirements and patterns of data and data sizes and computer hardware will benefit from or leverage different aspects of these variants.

This term we've already discussed / implemented a number of data structure variants:

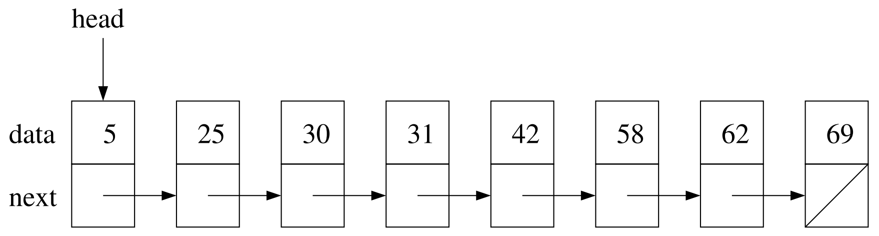
- single vs. doubly linked lists
  using more memory can improve convenience and running time for key operations
- dummy nodes or circular linked lists – *can reduce need for special case / corner case code*
- unrolled linked list (Homework 5)
- stack and queue – *restricted/reduced(!) set of operations on array/vector and list*
- rope – *string representation that allows for fast insert (split, concat, concat) and erase (split, split, concat)*
- hash table (Homework 9): separate chaining vs open addressing – *reduce memory and avoid pointer dereferencing*

We'll discuss just a few additional variants today. The list below is certainly not comprehensive!

- skip list
- red-black tree
- B+ tree
- quad tree
- leftist heap (if we have time)

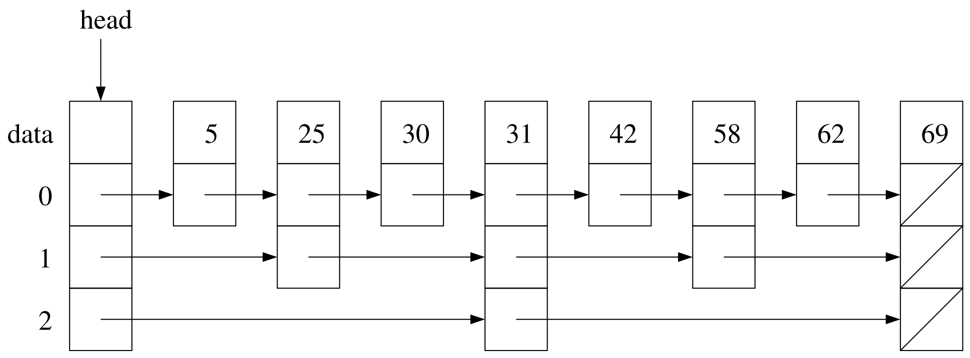
26.3 Skip List - Overview

- Consider a classic singly-linked list storing a collection of n integers in sorted order.



- If we want to check to see if '42' is in the list, we will have to linearly scan through the structure, with $O(n)$ running time.

- Even though we know the data is sorted... The problem is that unlike an array / vector, we can't quickly jump to the middle of a linked list to perform a binary search.
- What if instead we stored a additional pointers to be able to jump to the middle of the chain? A skip list stores sorted data with multiple levels of linked lists. Each level contains roughly half the nodes of the previous level, approximately every other node from the previous level.



- Now, to find / search for a specific element, we start at the highest level (level 2 in this example), and ask if the element is before or after each element in that chain. Since it's after '31', we start at node '31' in the next lowest level (level 1). '42' is after '31', but before '58', so we start at node '31' in the next lowest level (level 0). And then scanning forward we find '42' and return 'true' = yes, the query element is in the structure.

26.4 Skip List - Discussion

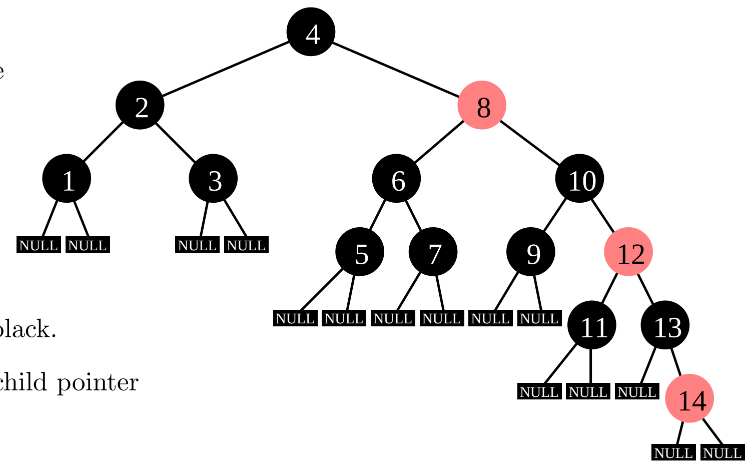
- How are elements inserted & erased? (Once the location is found) Just edit the chain at each level.
- But how do we determine what nodes go at each level? Upon insertion, generate a top level for that element at random (from [0,log n] where n is the # of elements currently in the list ... *details omitted!*)
- The overall hierarchy of a skip list is similar to a binary search tree. Both a skip list and a binary search tree work best when the data is balanced.

  Draw an (approximately) balanced binary search tree with the data above. How much total memory does the skip list use vs. the BST? Be sure to count all pointers – and don't forget the parent pointers!

- What is the height of a skip list storing n elements? What is the running time for `find`, `insert`, and `erase` in a skip list?

- Compared to BSTs, in practice, *balanced* skip lists are simpler to implement, faster (same order notation, but smaller coefficient), require less total memory, and work better in parallel. Or maybe they are similar...

26.5 Red-Black Trees



In addition to the binary search tree properties, the following red-black tree properties are maintained throughout all modifications to the data structure:

1. Each node is either red or black.
2. The NULL child pointers are black.
3. Both children of every red node are black. Thus, the parent of a red node must also be black.
4. All paths from a particular node to a NULL child pointer contain the same number of black nodes.

What tree does our `ds_set` implementation produce if we insert the numbers 1-14 *in order*?
The tree at the right is the result using a red-black tree. Notice how the tree is still quite balanced.
Visit these links for an animation of the sequential insertion and re-balancing:

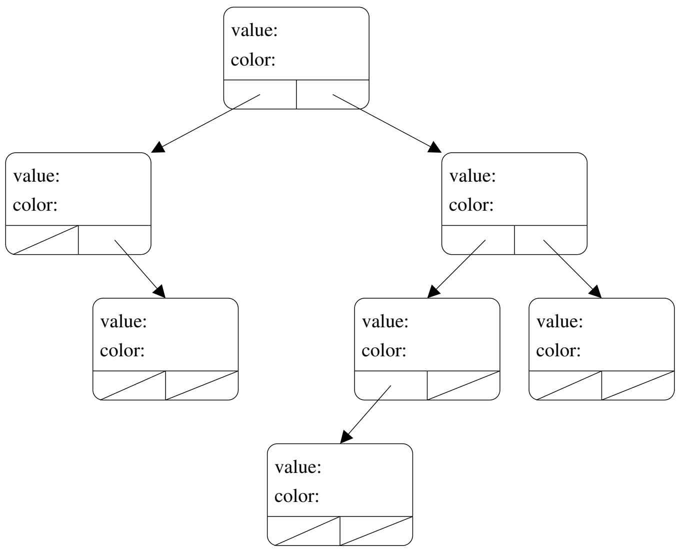
http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/RedBlackTree-Example.html
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
http://www.youtube.com/watch?v=vDHFF4wjWYU&noredirect=1

- What is the best/average/worst case height of a red-black tree with n nodes?

- What is the best/average/worst case shortest-path from root to leaf node in a red-black tree with n nodes?

26.6 Exercise

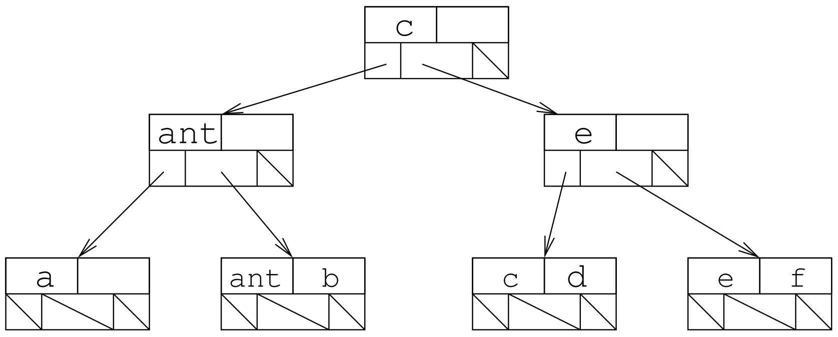
Fill in the tree on the right with the integers 1-7 to make a binary search tree. Also, color each node "red" or "black" so that the tree also fulfills the requirements of a Red-Black tree.



Draw two other red-black binary search trees with the values 1-7.

26.7 B+ Trees

- Unlike binary search trees, nodes in B+ trees (and their predecessor, the B tree) have up to b children. Thus B+ trees are very flat and very wide. This is good when it is very expensive to move from one node to another.
- B+ trees are supposed to be associative (i.e. they have key-value pairs), but we will just focus on the keys for now. *Question: Where would the values be stored?*
- Just like STL `map` and STL `set`, these *keys* and *values* can be any type, but *keys* must have an `operator<` defined.
- In a B tree value-key pairs can show up anywhere in the tree, in a B+ tree all the key-value pairs are in the leaves and the non-leaf nodes contain duplicates of some keys.
- In either type of tree, all leaves are the same distance from the root.
- The keys are always sorted in a B/B+ tree node, and there are up to $b-1$ of them. They act like $b-1$ binary search tree nodes mashed together.
- In fact, with the exception of the root, nodes will always have between roughly $\frac{b}{2}$ and $b-1$ keys (in our implementation).
- If a B+ tree node has k keys $key_0, key_1, key_2, \ldots, key_k$, it will have $k+1$ children. The keys in the leftmost child must be $< key_0$, the next child must have keys such that they are $\geq key_0$ and $< key_1$, and so on up to the rightmost child which has only keys $\geq key_k$.
- Example of a B+ Tree:



- Note: "a" will come before "ant" lexicographically, in other words "a" < "ant"
- Considerations in a full implementation:
  - What happens when we want to add a key to a node that's already full?
  - How do we remove values from a node?
  - How do we ensure the tree stays balanced?
  - How to keep leaves linked together? Why would we want this?
  - How to represent key-value pairs?

Exercise: Consider a BPlusTree node with $b = 5$ and keys: "ant", "bear", "bee", "cod".

If a search is looking for the key "cat" and the node is not a leaf, counting from the left, which child should the search use?

Exercise: Consider a BPlusTree node with b children per node. What is the maximum number of nodes that can be on level i?
(assuming the root is on level 0)

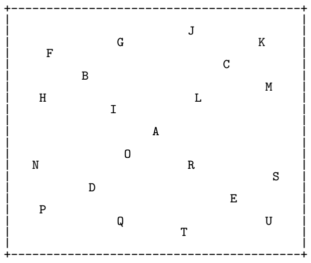
Exercise: Draw a B+ tree with $b = 3$ with values inserted in the order $1, 2, 3, 4, 5, 6$. Now draw a B+ tree with $b = 3$ and values inserted in the order $6, 5, 4, 3, 2, 1$. Hint: The two trees have a different number of levels. https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

26.8 Quad Tree - Overview

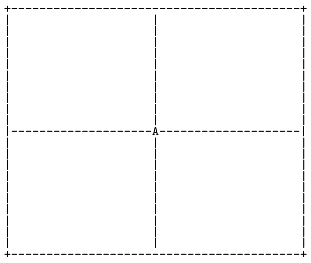
The *quad tree* data structure is a 2D generalization of the 1-dimensional binary search tree. The 3D version is called an *octree*, or in higher dimensions it is called a *k-d tree.* These structures are used to improve the performance of applications that use large spatial data sets including: ray tracing in computer graphics, collision detection for simulation and gaming, motion planning for robotics, nearest neighbor calculation, and image processing.

The diagrams below illustrate the incremental construction of a quad tree. We add the 21 *two-dimensional points* shown in the first image to the tree structure. We will add them in the alphabetical order of their letter *labels.* Each time a point is added we locate the rectangular region containing that point and subdivide that region into 4 smaller rectangles using the *x,y* coordinates of that point as the vertical and horizontal dividing lines.

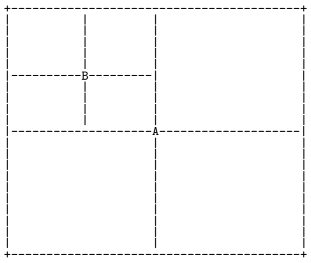
input points



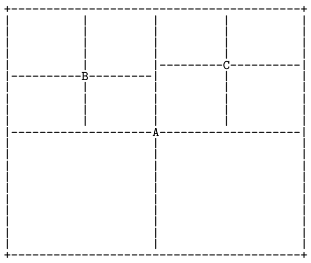
after adding the 1st point



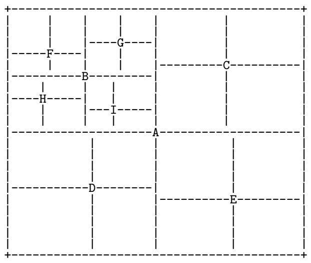
after adding the 2nd point



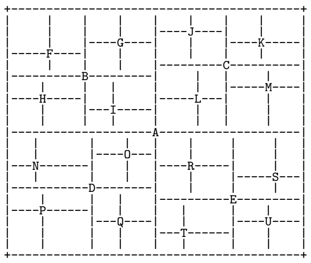
after adding the 3rd point



after adding 9 points



after adding all 21 points



Each node in the structure has 4 children. (Or 8 children if we're making a 3-dimensional octree). Here's a 'sideways' printing of the finished tree structure from the example above:

```
A (20,10)
    B (10,5)
        F (5,3)
        G (15,2)
        H (4,7)
        I (14,8)
    C (30,4)
        J (25,1)
        K (35,2)
        L (26,7)
        M (36,6)
    D (11,15)
        N (3,13)
        O (16,12)
        P (4,17)
        Q (15,18)
    E (31,16)
        R (25,13)
        S (37,14)
        T (24,19)
        U (36,18)
```

26.9 Quad Tree - Discussion

- How does the order of point insertion affect the constructed tree?
  What if we inserted the point labeled 'B' first, and the point labeled 'A' second?

- Can we easily erase an item from the quad tree? What if it's not a leaf node?

- Alternately (in fact, more typically), the quad tree / octree / k-d tree may simply split at the midpoint in each dimension. In this way the intermediate tree nodes don't store a data point. All data points are stored only at the leaves of the tree.

26.10 Leftist Heaps — Overview

- Our goal is to be able to merge two heaps in $O(\log n)$ time, where n is the number of values stored in the larger of the two heaps.
  - Merging two binary heaps (where every row but possibly the last is full) requires $O(n)$ time
- Leftist heaps are binary trees where we deliberately attempt to eliminate any balance.
  - Why? Well, consider the most *unbalanced* tree structure possible. If the data also maintains the heap property, we essentially have a sorted linked list.
- Leftists heaps are implemented explicitly as trees (rather than vectors).

26.11 Leftist Heaps — Mathematical Background

- **Definition:** The *null path length* (NPL) of a tree node is the length of the shortest path to a node with 0 children or 1 child. The NPL of a leaf is 0. The NPL of a NULL pointer is -1.
- **Definition:** A *leftist tree* is a binary tree where at each node the null path length of the left child is greater than or equal to the null path length of the right child.
- **Definition:** The *right path* of a node (e.g. the root) is obtained by following right children until a NULL child is reached. In a leftist tree, the right path of a node is at least as short as any other path to a NULL child. The right child of each node has the lower null path length.
- **Theorem:** A leftist tree with $r > 0$ nodes on its right path has at least $2^r - 1$ nodes.
  - This can be proven by induction on r.
- **Corollary:** A leftist tree with n nodes has a right path length of at most $\lfloor \log(n+1) \rfloor = O(\log n)$ nodes.
- **Definition:** A *leftist heap* is a leftist tree where the value stored at any node is less than or equal to the value stored at either of its children.

26.12 Leftist Heap Operations

- The push/insert and pop/delete_min operations will depend on the merge operation.
- Here is the fundamental idea behind the merge operation. Given two leftist heaps, with h1 and h2 pointers to their root nodes, and suppose h1->value <= h2->value. Recursively merge h1->right with h2, making the resulting heap h1->right.
- When the leftist property is violated at a tree node involved in the merge, the left and right children of this node are swapped. This is enough to guarantee the leftist property of the resulting tree.
- Merge requires $O(\log n + \log m)$ time, where m and n are the numbers of nodes stored in the two heaps, because it works on the right path at all times.

26.13 Leftist Heap Implementation

- Our Node class:

```
template <class T> class LeftNode {
public:
  LeftNode() : npl(0), left(0), right(0) {}
  LeftNode(const T& init) : value(init), npl(0), left(0), right(0) {}
  T value;
  int npl; // the null-path length
  LeftNode* left;
  LeftNode* right;
};
```

- Here are the two functions used to implement leftist heap merge operations. Function merge is the driver. Function merge_helper does most of the work. These functions call each other recursively.

```
template <class T>
LeftNode<T>* merge(LeftNode<T> *H1,LeftNode<T> *H2) {
  if (!h1)
    return h2;
  else if (!h2)
    return h1;
  else if (h2->value > h1->value)
    return merge_helper(h1, h2);
  else
    return merge_helper(h2, h1);
}

template <class T>
LeftNode<T>* merge_helper(LeftNode<T> *h1, LeftNode<T> *h2) {
  if (h1->left == NULL)
    h1->left = h2;
  else {
    h1->right = merge(h1->right, h2);
    if(h1->left->npl < h1->right->npl)
      swap(h1->left, h1->right);
    h1->npl = h1->right->npl + 1;
  }
  return h1;
}
```

26.14 Leftist Heap Exercises

1. Explain how merge can be used to implement insert and delete_min, and then write code to do so.

2. Show the state of a leftist heap at the end of:

```
insert 1, 2, 3, 4, 5, 6
delete_min
insert 7, 8
delete_min
delete_min
```