CSCI-1200 Data Structures — Fall 2024
Lecture 23 – Hybrid / Variant Data Structures

Review from Lecture 22

- STL Queue and STL Stack, What's a Priority Queue?
- Definition of a Binary Heap, A Priority Queue as a Heap
- Implementing Pop (Delete Min) and Push (Insert), A Heap as a Vector
- Heapify – Building a Heap (all at once), Heap Sort, Complexity Analysis

Today's Lecture

- Finish material on Priority Queues from Lecture 22 and Lab 13
- Some variants on the classic data structures...

22.12 Heap Operations Time Complexity Analysis

- Both `percolate_down` and `percolate_up` are $O(\log n)$ in the worst-case. Why?
  - The item we temporarily place at the root after a pop was taken from the bottom row and willl probably navigate the height of the tree before stopping.
  - The new item we push and initially place in the first open spot in the last row *might* be the new smallest value item and belong at the root.
- But, `percolate_up` (and as a result `push`) is $O(1)$ in the average case. Why?
  - *IF* the items are inserted in random order, most items will not be smaller than all previously inserted items.
  - In fact, the newly inserted item has a 50% chance of being in the bottom half by value of all items inserted so far. Thus it has a 50% chance of belonging in the bottom row of the tree (bottom row of the tree holds half of the items).
  - 50% chance of needing 0 swaps, 25% chance of needing 1 swap, 12% chance of 2 swaps, etc.

22.13 Building A Heap ... starting with all of your data in an unorganized vector

- In order to build a heap from a unorganized vector of values, for each index from $\lfloor (n-1)/2 \rfloor$ down to 0, run `percolate_down`. Show that this fully organizes the data as a heap and requires at most $O(n)$ operations.
- If instead, we ran `percolate_up` from each index starting at index 0 through index n-1, we would get properly organized heap data, but incur a $O(n \log n)$ cost in the worst case. Why?

22.14 Heap Sort - Implemented in Lab 13

- Heap Sort is a simple algorithm to sort a vector of values: Build a heap and then run n consecutive `pop` operations, storing each "popped" value in a new vector.
- It is straightforward to show that this requires $O(n \log n)$ time.
- **Exercise:** Implement an *in-place* heap sort. An in-place algorithm uses only the memory holding the input data – a separate large temporary vector is not needed. Side goal: Keep the number of element copy/swap operations to a minimum!

22.15 Summary Notes about Vector-Based Priority Queues

- Priority queues are conceptually similar to queues, but the order in which values / entries are removed ("popped") depends on a priority.
- Heaps, which are simply drawn with a binary tree but are implemented in a vector, are the data structure of choice for a priority queue.
- In some applications, the priority of an entry may change while the entry is in the priority queue. This requires that there be "hooks" (usually in the form of indices) into the internal structure of the priority queue. This is an implementation detail we have not discussed.

22.16 Review: Vector vs. List vs. BST vs Heap: Time Complexity Analysis

	find smallest value	remove smallest value	insert value
unsorted vector/array			
sorted vector/array			
unsorted linked list			
sorted linked list			
binary search tree (balanced)			
binary heap			

23.1 The Basic Data Structures

This term we've studying the details of a spectrum 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 (Lectures 20 & 21, Lab 12, Homework 9)
- binary heap / priority queue (Lecture 22, Lab 13, Homework 10)

23.2 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*
- compressing a sparse matrix/grid to save storage space (HW3 jagged array)
- 2D arrays/vectors (HW5) or 2D linked grid/matrix (various practice problems)
- BVH spatial data structure (HW8) – *organizing 2D/3D geometric data*
- red-black tree – *an algorithm to automatically balance a binary search tree*
- hash table: separate chaining vs open addressing (HW9) – *reduce memory and avoid pointer dereferencing to improve performance*
- stack and queue – *restricted/reduced(!) set of operations on array/vector and list*

In the final homework we will see a variant on the priority queue:

- priority queue with backpointers (HW10) – *when you need to update data already in the structure*

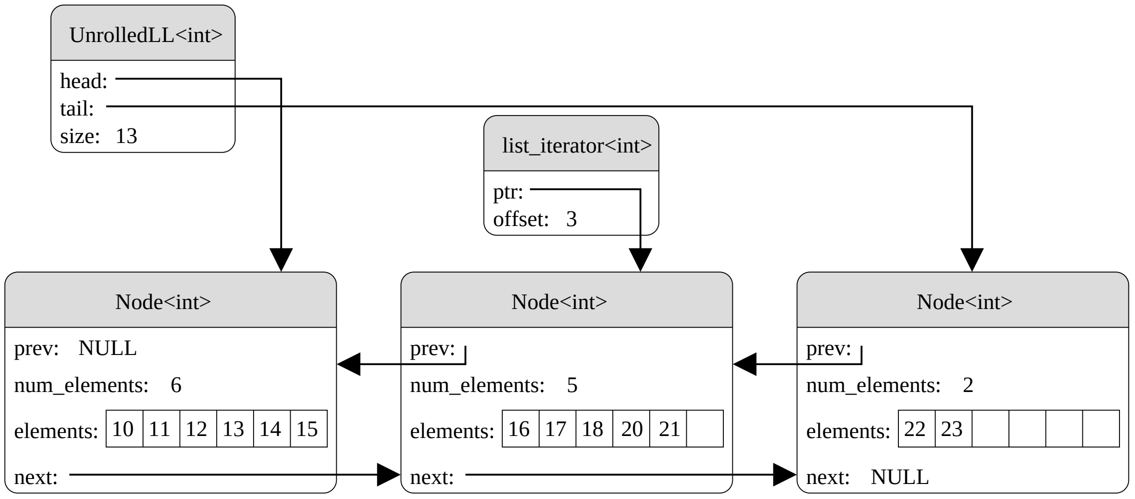
We'll discuss just a few additional variants today.

- unrolled linked list
- skip list
- quad tree
- trie (a.k.a. prefix tree)
- suffix tree

The list above is just a sampling of the possible variety of hybrid / variant data structures!

23.3 Unrolled Linked List - Overview

- An *unrolled linked list* data structure is a hybrid of an array / vector and a linked list. It is very similar to a standard doubly linked list, except that *more than one element* may be stored at each node.
- This data structure can have performance advantages (both in memory and running time) over a standard linked list when storing small items and can be used to better align data in the cache.
- Here's a diagram of an unrolled linked list:



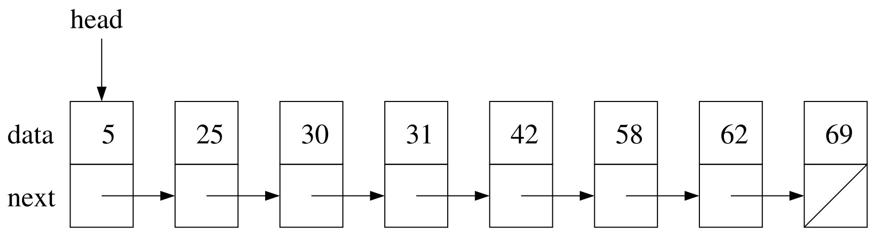
- Each `Node` object contains a *fixed size* array (size = 6 in the above example) that will store 1 or more elements from the list. The elements are ordered from left to right.
- From the outside, this unrolled linked list should perform exactly like an STL list containing the numbers 10 through 23 in sorted order, except we've just erased '19'. Note that to match the behavior, the `list_iterator` object must also change. The iterator must keep track of not only which `Node` it refers to, but also which element within the `Node` it's on. This can be done with a simple offset index. In the above example, the iterator refers to the element "20".
- Just like regular linked lists, the unrolled linked list supports speedy `insert` and `erase` operations in the middle of the list. The diagram above illustrates that after erasing an item it is often more efficient to store one fewer item in the affected `Node` than to shift *all* elements (like we have to with an array/vector).
- And when we insert an item in the middle, we might need to splice a new `Node` into the chain if the current `Node` is "full" (there's not an empty slot).

23.4 Unrolled Linked List - Discussion

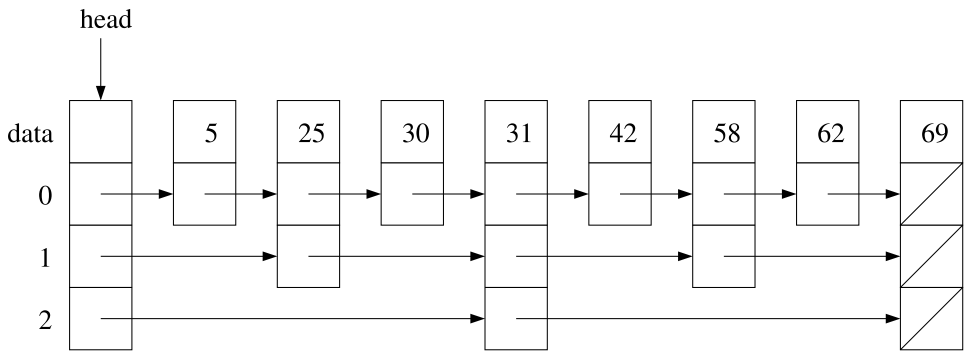
- Say that `Foo` is a custom C++ class that requires 16 bytes of memory. If we create a basic doubly-linked list of n `Foo` objects on a 64 bit machine, how much total memory will we use? Assume that each blob of memory allocated on the heap has an 8 byte header.
- Now instead, let's store n booleans in a basic doubly-linked list. How much total memory will that use? Assume that heap allocations must round up to the nearest 8 byte total size.
- Finally, let's instead use an unrolled linked list. How many boolean values items should we store per `Node`? Call that number k. How much total memory will we use to store n booleans? What if the nodes are all 100% "full"? What if the nodes are on average 50% "full"?

23.5 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.

23.6 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?

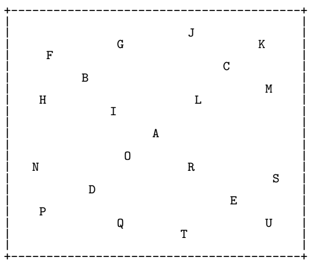
- Compared to BSTs, in practice, *balanced* skip lists are simpler to implement, faster (same order notation, but smaller coefficient), require less total memory, and work better in parallel. Or maybe they are similar...

23.7 Quad Tree - Overview

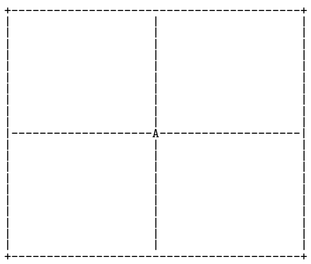
The *quad tree* data structure is a 2D generalization of the 1-dimensional binary search tree. The 3D version is called an *octree*, or in higher dimensions it is called a *k-d tree*. These structures are used to improve the performance of applications that use large spatial data sets including: ray 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.

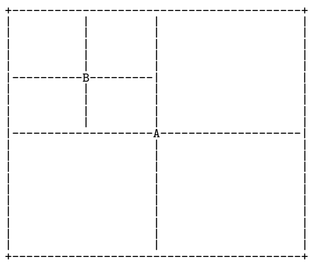
input points



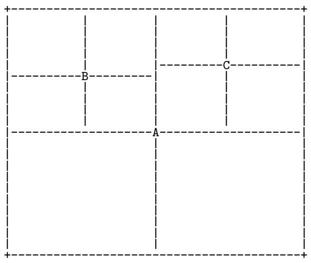
after adding the 1st point



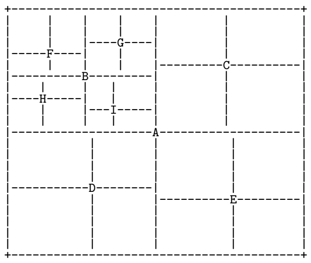
after adding the 2nd point



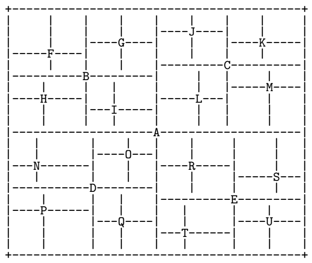
after adding the 3rd point



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)
```

23.8 Quad Tree - Discussion

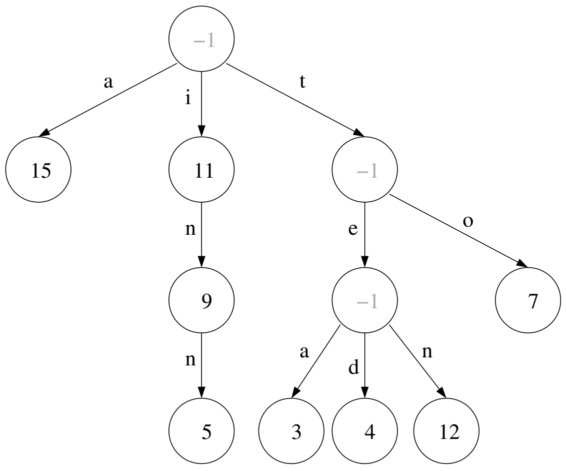
- How does the order of point insertion affect the constructed tree?
  What if we inserted the point labeled 'B' first, and the point labeled 'A' second?

- 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.

23.9 Trie / Prefix Tree - Overview

- Next up, let's look at alternate to a map or hash map for storing *key* strings and an associated *value* type. *NOTE: We'll cover the classic hash table in lecture next week!*
- In a trie or prefix tree, the key is defined not by storing the data at the node or leaf, but instead by the path of to get to that node. Each *edge* from the root node stores one character of the string. The node stores the value for the key (or NULL or a special value, e.g., '-1', if the path to that point is not a valid key in the structure).



key	value
a	15
i	11
in	9
inn	5
tea	3
ted	4
ten	12
to	7

- Lookup in the structure is fast, O(m) where m is the length (# of characters) in the string. A hash table has similar lookup (since we have to hash the string which generally involves looking at every letter). If $m << n$, we can say this is O(1).

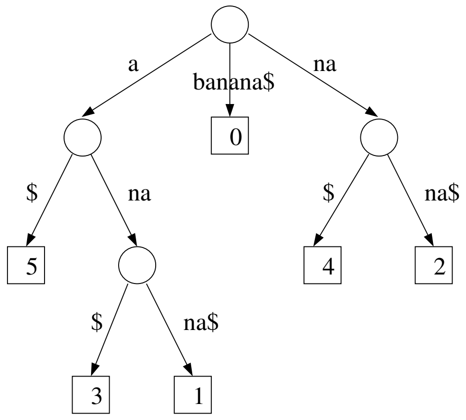
23.10 Trie / Prefix Tree - Discussion

- What is the worst case # of children for a single node? What are the member variables for the `Node` class?
- Unlike a hash table, we can iterate over the keys in a trie / prefix tree in sorted order.
  Exercise: Implement the trie sorted-order iterator (in code or pseudocode) and print the table on the right.

23.11 Suffix Tree - A Brief Introduction...

- Instead of only encoding the complete string when walking from root to leaf... let's store every possible substring of the input.
- This toy example stores 'banana', and all suffix substrings of 'banana'. Each leaf node stores the start position of the substring within the original string. The '$' character is a special terminal character.
- Suffix trees clearly require much more memory than other data structures to store the input string, but do so to gain performance on certain operations....

  Suffix trees help us efficiently find the longest common substring – *in linear time*. This is an important problem in genome sequencing and computational biology.
- Clever algorithms have been developed to efficiently construct suffix trees.



... and we're certainly out of time for today. There are many more wonderful data structures to explore. This semester you have learned the tools to study new structures, compare and contrast operation efficiency and memory usage of different structures, and to develop your own data structures for specific applications.