CSCI-1200 Data Structures — Fall 2019
Lecture 21 – Priority Queues II
+ Bonus Data Structures I

Review from Lecture 20

- 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)

Today's Lecture

- A Heap as a Vector
- Building a Heap
- Heap Sort
- Comparison: Vector vs. List vs. BST vs. Heap – Time Complexity Analysis
- Bonus Data Structures (Part 1)
  - Merging heaps are the motivation for *leftist heaps*
  - Unrolled Linked List
  - Skip List

21.1 Implementing a Heap with a Vector (instead of Nodes & Pointers)

- In the vector implementation, the tree is never explicitly constructed. Instead the heap is stored as a vector, and the child and parent "pointers" can be implicitly calculated.
- To do this, number the nodes in the tree starting with 0 first by level (top to bottom) and then scanning across each row (left to right). These are the vector indices. Place the values in a vector in this order.
- As a result, for each subscript, i,
  - The parent, if it exists, is at location $\lfloor (i-1)/2 \rfloor$.
  - The left child, if it exists, is at location $2i+1$.
  - The right child, if it exists, is at location $2i+2$.
- For a binary heap containing n values, the last leaf is at location $n-1$ in the vector and the last internal (non-leaf) node is at location $\lfloor (n-1)/2 \rfloor$.
- The standard library (STL) `priority_queue` is implemented as a binary heap.

21.2 Heap as a Vector Exercises

- Draw a binary heap with values: 52 13 48 7 32 40 18 25 4, first as a tree of nodes & pointers, then in vector representation.

- Starting with an initially empty heap, show the vector contents for the binary heap after each `delete_min` operation.

```
push 8, push 12, push 7, push 5, push 17, push 1,
pop,
push 6, push 22, push 14, push 9,
pop,
pop,
```

21.3 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. Why?

21.4 Heap Sort

- 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!

21.5 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 conceptually 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.

21.6 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			

21.7 Leftist Heaps — Overview

- Merging two classic binary heaps (where every row but possibly the last is full) requires $O(n)$ time.
- Leftist heaps are *deliberately unbalanced heaps* (in the limit, they are essentially a sorted linked list), with the goal to allow merging of two heaps in $O(\log n)$ time, where n is the number of values stored in the larger heap.
- Leftists heaps are implemented explicitly as trees (rather than vectors).
- The *null path length* (NPL) 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.
- 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.
- 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.
- 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. A leftist tree with n nodes has a right path length of at most $\lfloor \log(n+1) \rfloor = O(\log n)$ nodes.

21.8 Leftist Heap — Implementation

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

template <class T>  LeftNode<T>* merge(LeftNode<T> *H1, LeftNode<T> *H2) {
  if      (h1 == NULL)            { return h2; }
  else if (h2 == NULL)            { 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;
}
```

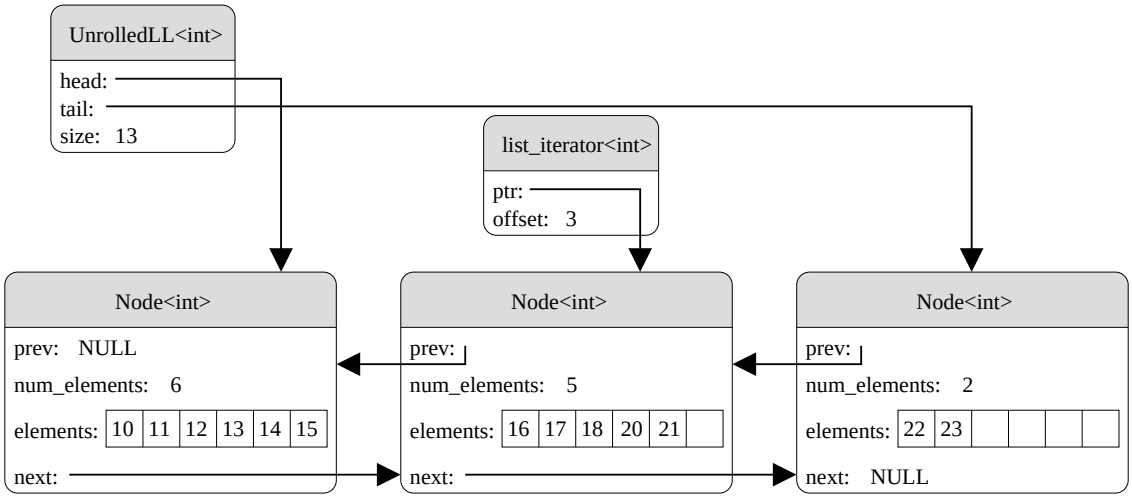
21.9 Leftist Heap — Discussion & Exercises

- `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.
- **Exercise:** Finish the implementation: `insert` and `delete_min` can be implemented using `merge`.
- **Exercise:** 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
```

21.10 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).

21.11 Unrolled Linked List - Discussion

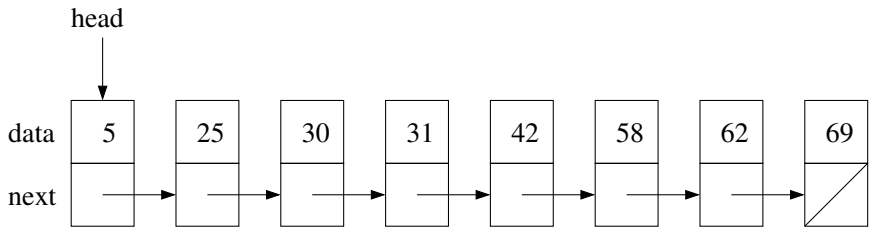
- Say that `Foo` is a custom C++ class that requires 16 bytes of memory. If we create a basic doubly-linked list of n `Foo` objects on a 64 bit machine, how much total memory will we use? Assume that each blob of memory allocated on the heap has an 8 byte header.

- 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.

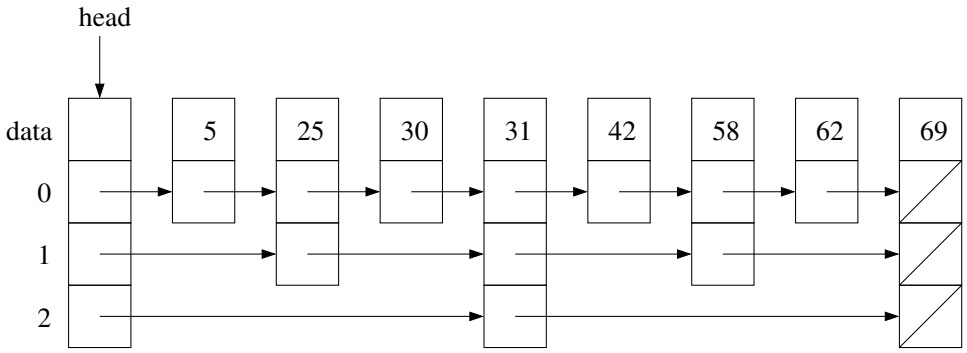
- Finally, let's instead use an unrolled linked list. How many boolean values items should we store per `Node`? Call that number k. How much total memory will we use to store n booleans? What if the nodes are all 100% "full"? What if the nodes are on average 50% "full"?

21.12 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.

21.13 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...