CSCI-1200 Data Structures — Fall 2021
Lecture 21 – Hybrid / Variant Data Structures

Announcements: Test 3 Information

- Test 3 will be held Thursday, November 18th from 6-7:50pm.
- Your exam room & zone assignment will be posted on Submitty.
  Note: We will re-shuffle the room & zone assignments from Exams 1 & 2.
- Coverage: Lectures 1-22, Labs 1-12, HW 1-8.
- Studying
  - Practice problems from previous tests are available on the course website.
  - Sample solutions to the practice problems will be posted on Wednesday morning.
  - The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
  - You should practice timing yourself as well. The test will be 110 minutes and there will be 100 points. If a problem is worth 25 points, budgeting 25 minutes for yourself to solve the problem is a good time management technique.
  - OPTIONAL: Prepare a 2 page, black & white, 8.5x11", portrait orientation .pdf of notes you would like to have during the test. This may be digitally prepared or handwritten and scanned or photographed. The file may be no bigger than 2MB. You will upload this file to Submitty gradeable "Test 3 Notes Page (Optional)" before Wednesday November 17th @11:59pm. We will print this and attach it to your test. No other notes may be used during the test.
  - Going in to the test, you should know what big topics will be covered on the test. As you skim through the problems, see if you can match up those big topics to each question. Even if you are stumped about how to solve the whole problem, or some of the details of the problem, make sure you demonstrate your understanding of the big topic that is covered in that question.
  - Re-read the problem statement carefully. Make sure you didn't miss anything.
- Bring to the test room:
  - Your mask. Masks must be worn 100% of the time during the exam.
    Please properly wear a well fitted surgical mask or N95 or KN95 mask to the exam room.
    We will have surgical masks at the front of the room if you do not already have one.
  - NOTE: Please use the restroom before entering the exam room.
    Students must remain in their seats until they are ready to turn in their exam.
  - Your Rensselaer photo ID card. We will be checking IDs when you turn in your exam.
  - Pencil(s) & eraser (pens are ok, but not recommended). The test *will* involve handwriting code on paper, short answer problem solving, and may require you to draw a memory diagram. Neat legible handwriting is appreciated. We will be somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)
  - Do not bring your own scratch paper. The exam packet will include sufficient scratch paper.
  - Computers, cell-phones, smart watches, calculators, music players, etc. are not permitted. Please do not bring your laptop, books, backpack, etc. to the test room – leave everything in your dorm room. *Unless you are coming directly from another class or sports/club meeting.*

Review from Lecture 20

- Some more practice exercises with trees & Big O Notation
- Implement `erase` from a `ds_set`
- Limitations of our `ds_set` implementation, brief intro to red-black trees

Today's Lecture

- Some variants on the classic data structures...

21.1 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 22 & 23, Lab 12, Homework 9)
- binary heap / priority queue (Lecture 25, Lab 13, Homework 10)

21.2 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*
- 2D arrays/vectors (HW3) or 2D linked grid/matrix (HW5)
- quad trees (Homework 8) – *good for 2D spatial data, in 3D we use an octree*
- red-black tree – *an algorithm to automatically balance a binary search tree*

In the remaining lecture & homeworks we'll cover 2 additional classic data structures and several more variants...

- hash table: separate chaining vs open addressing – *reduce memory and avoid pointer dereferencing*
- stack and queue – *restricted/reduced(!) set of operations on array/vector and list*
- priority queue with backpointers (may be used in Homework 10) – *when you need to update data already in the structure*
- leftist heap (might mention this in Lecture 25...)

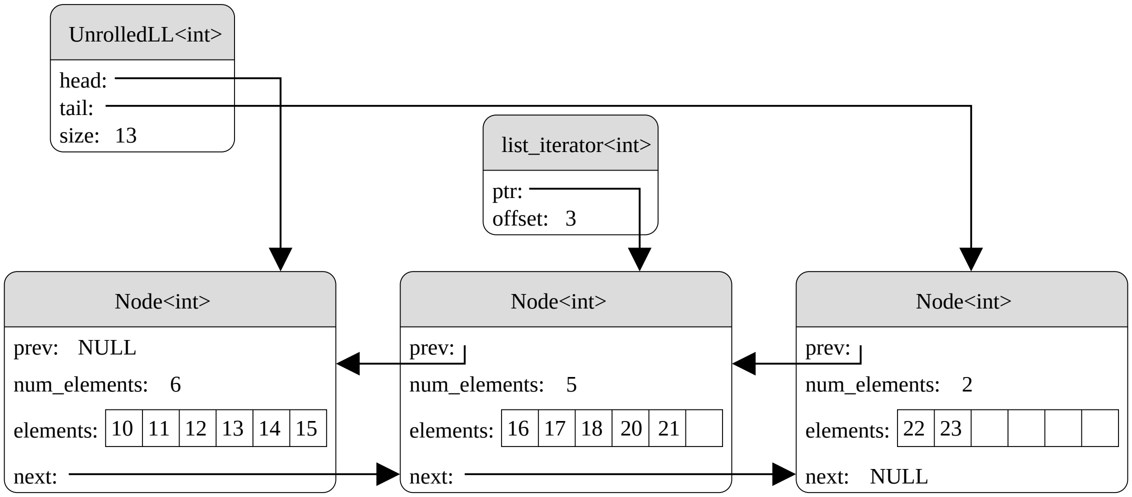
We'll discuss just a few additional variants today.

- unrolled linked list
- skip list
- trie (a.k.a. prefix tree)
- suffix tree
- bounding volume hierarchy

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

21.3 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:



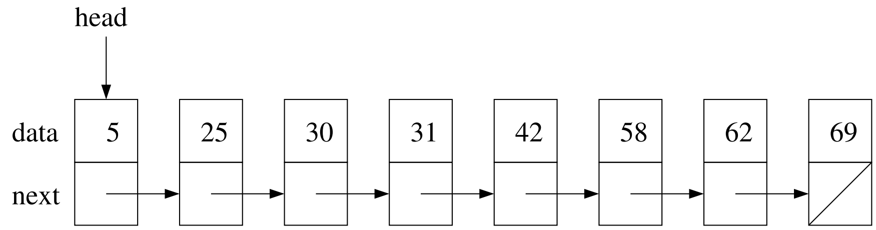
- Each `Node` object contains a *fixed size* array (size = 6 in the above example) that will store 1 or more elements from the list. The elements are ordered from left to right.
- From the outside, this unrolled linked list should perform exactly like an STL list containing the numbers 10 through 23 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).

21.4 Unrolled Linked List - Discussion

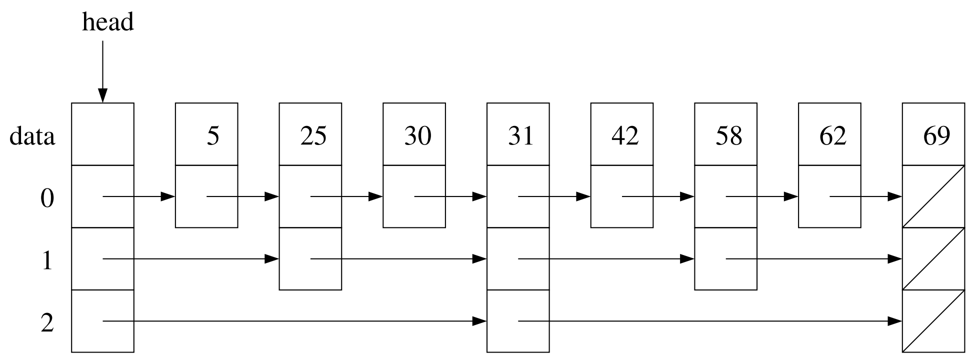
- Say that `Foo` is a custom C++ class that requires 16 bytes of memory. If we create a basic doubly-linked list of n `Foo` objects on a 64 bit machine, how much total memory will we use? Assume that each blob of memory allocated on the heap has an 8 byte header.
- Now instead, let's store n booleans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

21.5 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.

21.6 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.

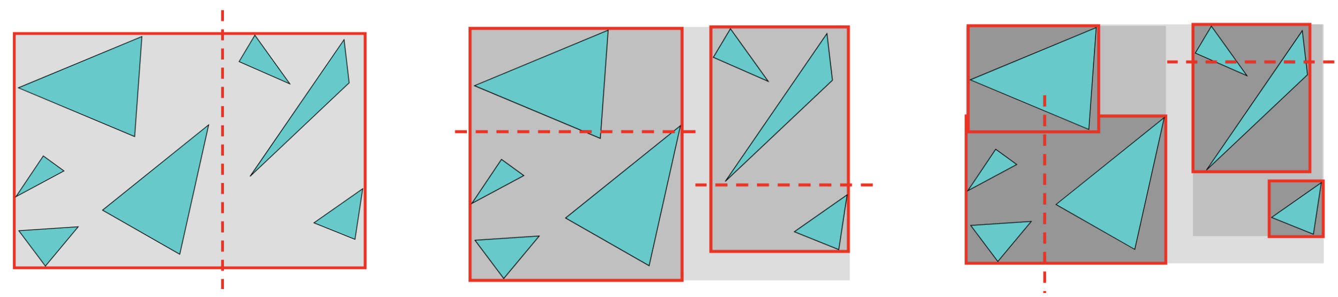
  Draw an (approximately) balanced binary search tree with the data above. How much total memory does the skip list use vs. the BST? Be sure to count all pointers – and don't forget the parent pointers!

- 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...

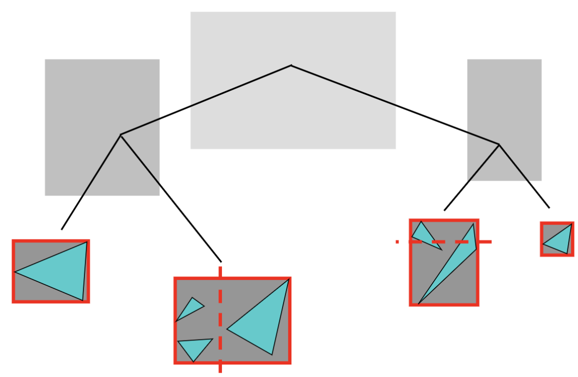
21.7 Bounding Volume Hierarchy - Overview

The *bounding volume hierarchy* (BVH) is a 2D/3D spatial data structure – like the *quad tree* from Homework 8. While both can store points, the BVH excels at storing geometric shapes such as disks, triangles, and quadrilaterals. A BVH facilitates efficient spatial queries for a variety of applications, 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 construction of a BVH to store a collection of 7 triangles. Each node in the tree stores the coordinates of the smallest rectangular box that contains all of the objects in the tree beneath it – *the bounding box* of those objects. Sometimes *bounding spheres* are used instead of boxes!



At each node the objects are separated into two groups, often selected by a cutting line on the x or y axes. Two subtrees of objects are created and the construction process recurses.



The BVH is most effective at accelerating geometric operations if the bounding boxes of the two subtrees are both significantly smaller than the parent node – e.g., approximately half the volume. However, because the objects being stored in the tree can be large shapes, it is not uncommon that bounding boxes of the subtrees *overlap*, which will proportionally decrease the effectiveness/usefulness of the BVH.

21.8 Bounding Volume Hierarchy - Discussion

- Starting at the root node, with the initial collection of geometric shapes and their bounding box, how should we choose to separate them into two groups?

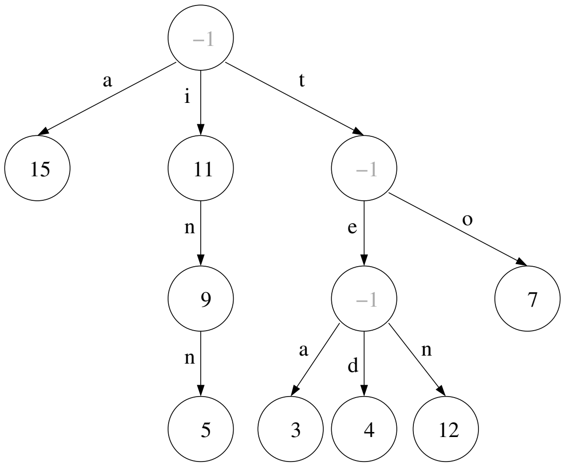
- Should we prioritize separating into two groups whose bounding boxes are equal volume? Or minimum volume? Or equal number of elements?

- Once a tree with many elements is constructed, can we add a new object? What edits to an existing tree are necessary to add the new object?

- Or should we start over and rebuild the whole BVH from scratch with the complete collection?

21.9 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).

21.10 Trie / Prefix Tree - Discussion

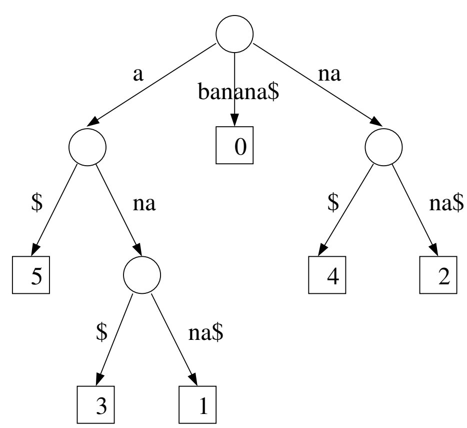
- What is the worst case # of children for a single node? What are the member variables for the `Node` class?

- 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.

21.11 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.