Basic Data Structures and Algorithms

Run time analysis: Big-O definition

- Suppose a program funning on input of size n has run time $f(n)$ seconds.
- Big-O gives an upper bound on run-time to within a constant factor A function $f(n)$ is said to be $O(g(n))$ if there exist constants C and N such that $f(n) < C \cdot g(n)$ for all $n > N$. (Draw picture.)
- Read "$f(n) = O(g(n))$" as "$f(n)$ is big-O of "g(n)."
- Here are the typical $g(n)$ functions in increasing order:
  - $g(n) = 1$, e.g. array lookup by index `i`
  - $g(n) = \log_2(n)$, e.g. binary search
  - $g(n) = n$, e.g. linear search
  - $g(n) = n \cdot \log_2(n)$, e.g. clever sorts
  - $g(n) = n^2$, e.g. selection sort
  - $g(n) = n^3$, e.g. matrix mulitplication (in $C = AB$, $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$)
  - $g(n) = 2^n$, e.g. traveling salesman
  - $g(n) = n!$, e.g. number of possible tic-tac-toe games
- Just reading a data set of size n is $O(n)$, so an $O(n)$ program usually counts as fast. Since $\log_2(n)$ is small for the n values we see, an $O(n \cdot \log_2(n))$ program is usually fast enough. Programs taking $O(n^2)$ or more time may work for small n but can be too slow for large n.
- Often the order of the algorithm usually matters a lot more than processor speed, coding skill, programming language, etc.

Basic data structures: preview

Here are the basic data structures we consider:

- unordered list
- sorted list
- stack
- queue
- binary tree (with a discussion of recursion as background)
- priority queue
- graph

In each case, we consider an array implementation and a linked implementation.

We also consider a few algorithms for searching:

- linear search
- binary search
- hashing

and sorting:

- insertion
- selection
- quick
- heap
- merge

Basic data structures

Here are details.

- Unordered list
  - array:
    * insert at `data[size]` and set `size = size + 1` (and allocate larger array and copy data over if `size == capacity`, which costs $O(\underline{\hspace{2cm}})$)
      e.g. insert L

      capacity: 7
      size: 4
      data:

J	T	N	D			

      $O(\underline{\hspace{2cm}})$
    * to remove `x`, find it at some index `i` and set `data[i] = data[size-1]` and `size = size - 1`
      e.g. remove T

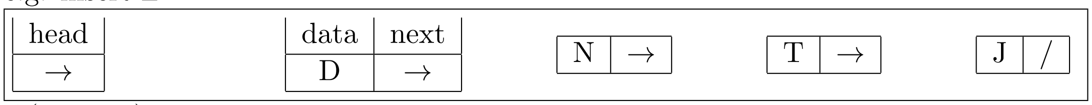
      capacity: 7
      size: 4
      data:

J	T	N	D			

      $O(\underline{\hspace{2cm}})$
  - linked via nodes of the form

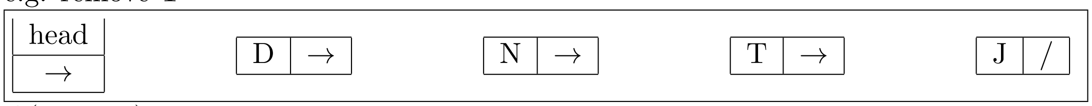
```
template <typename T>
struct Node {
    T data;
    Node<T> *next;
};
```

    * insert at head
      e.g. insert L



$O(\_\_\_\_\_\_)$

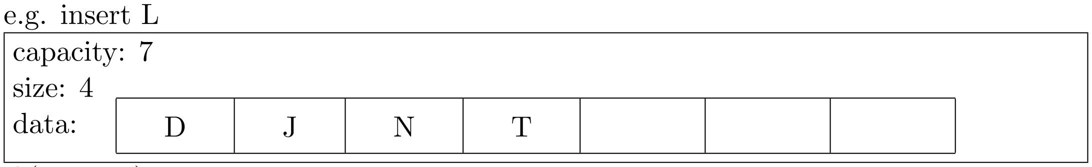
    * to remove `x`, set a reference `r` to the pointer to it and set `temp = r; r = r->next; delete temp`
      e.g. remove T



$O(\_\_\_\_\_\_)$

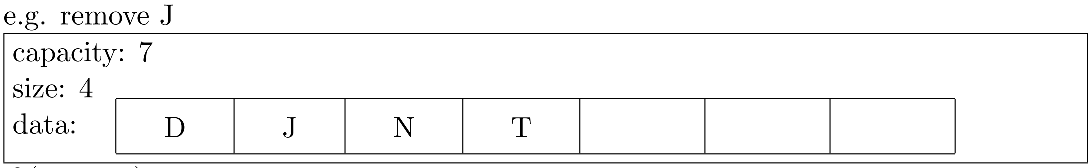
An unordered list can be uesd to hold a *set* of items.

- Sorted list (e.g. a list of student records sorted by name)
  - array
    * to insert `x`, find index `i` at which it belongs, shuffle each element right of `i` one to the right, and set `size = size + 1` (and allocate larger array and copy data over if `size == capacity`, which costs $O(\_\_\_\_\_\_)$)
      e.g. insert L



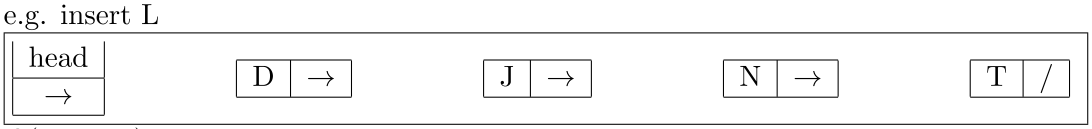
$O(\_\_\_\_\_\_)$

    * to remove `x`, find it at some index `i`, shuffle each element right of `i` one to the left, and set `size = size - 1`
      e.g. remove J



$O(\_\_\_\_\_\_)$

  - linked
    * to insert `x`, set a reference `r` to the pointer which should point to it and set `temp = new Node<T>; temp->data = x; temp->next = r; r = temp;`,
      e.g. insert L



$O(\_\_\_\_\_\_)$

- e.g. to remove `x`, set a reference `r` to the pointer which should point to it and set `temp = r; r = r->next; delete temp;`
  e.g. remove J

head				
→	D →	J →	N →	T /

  $O($_______$)$

- Stack

  A stack is a restricted-access list in which we insert (or *push*) only at the top and remove (or *pop*) only from the top. This is "last in, first out" (LIFO) behavior. The standard stack is a stack of cafeteria trays.

  – array

  - e.g. push E

    capacity: 7
    size: 4

data:	A	B	C	D			

    $O($_______$)$ (unless `size == capacity`, so we allocate larger array and copy data, which costs $O($_______$)$)

  - e.g. pop, returning _______

    capacity: 7
    size: 4

data:	A	B	C	D			

    $O($_______$)$ (unless we need

  – linked

  - insert at head
    e.g. push E

head				
→	D →	C →	B →	A /

    $O($_______$)$ (or $O($_______$)$ if reallocating array)

  - remove at head
    e.g. pop, returning _______

head				
→	D →	C →	B →	A /

    $O($_______$)$

  A stack facilitates deferring a task to return to it later. e.g.

  – Evaluate an infix arithmetic expression like `((3+4)*(1+2))` with a stack of operators. Evaluate a postfix expression like `3 4 + 1 2 + *` with a stack of operands.

  – Evaluate function calls via a stack containing parameters, local variables, and where to return for each active function call.

  – Parse a computer program with the help of a stack.

- Backtracking uses a stack of locations to facilitate orderly advancing and retreat from a dead end while searching for a destination. Two examples include the n-queens search and a depth-first search (DFS) of a graph (below).

- Queue

  A queue is a restricted-access list which allows insertions only at the back and deletions only at the front. This is "first in, first out" (FIFO) behavior. The standard queue is a check-out line at a grocery store.

  - array

    Use a *circular* array:

    * insert at `tail` (starting at 0), increment `tail`, wrapping via `% capacity`, and increment `size`
    * remove from `head` (starting at 0), increment `head`, also wrapping, and decrement `size`
    * e.g. insert J

      capacity: 7
      size: 5
      head: 4
      tail: 2
      data:

H	I			E	F	G

      $O(\_\_\_\_\_\_)$ (or $O(\_\_\_\_\_\_)$ if reallocating array)

    * e.g. remove, returning _______

      capacity: 7
      size: 5
      head: 4
      tail: 2
      data:

H	I			E	F	G

      $O(\_\_\_\_\_\_)$

  - linked

    * insert at tail

      e.g. insert J

head						tail
→	E →	F →	G →	H →	I /	←

      $O(\_\_\_\_\_\_)$

    * remove from head

      e.g. remove, returning _______

head						tail
→	E →	F →	G →	H →	I /	←

      $O(\_\_\_\_\_\_)$

  A queue facilitates processing tasks in the order they are received. e.g.

  - Schedule resources, like a computer processor, disk, or printer, for multiple users.
  - A buffering scheme can use a queue to save a fast device the trouble of waiting for a slow one: the fast one puts its request on the slow one's queue and moves on.
  - Smoothing with a moving average of N data points can use a queue to keep track of the current N points, inserting one and removing one at each step.
  - Breadth-first search (BFS) of a graph (below) uses a queue to store nodes to be processed.

- Trees, below, are recursive. Here is background on writing recursive functions.

  A *recursive* function solves a problem by solving one or more smaller instances of the same problem.

  Each recursive function needs a base case handled without recursion and a recursive call that reduces its arguments toward the base case. Here is a recipe for writing a recursive function:

  1. Write the prototype, e.g. "`void f(int n)`" or "`int f(Node *head)`" and a clear comment describing what `f(n)` or `f(head)` does.
  2. Write a case in which no recursive call is made. This is often `n == 0`, `n == 1`, or `head == 0` or `head->next = 0`.
  3. Assume the recursive call will work, e.g. that `f(n-1)` or `f(head->next)` will do what the comment on the prototype promises. (This is the hard part, requiring faith rather than effort.) Construct a solution around one or more recursive calls:
     - Consider doing work before recursive call.
     - Consider doing work after recursive call.
  4. Now, by a *miracle* (or by something like induction if you dislike miracles), `f()` works (e.g. for all `n >= 0`, or for all `head` pointers).

  e.g. Here are several functions to process a linked list.

```
// Returns the length of the list at which head points.
int length(Node *head) {
   if (!head) {                  // base case
      return 0;
   }
   return 1 + length(head->next) // solution from recursive call
}
```

  The order of statements inside a recursive function matters a lot. e.g.

```
// Prints the data from each Node of the list at which head points.
void print(Node *head) {
   if (head) { // The implicit base case is (head == 0).
      cout << head->data; // Do work on the way down the list.
      print(head->next);
   }
```

```
}

// Prints the data from each Node of the list at which head points
// in backward order
void print_backward(Node *head) {
   if (head) {
      print_backward(head->next);
      cout << head->data; // Do work on the way back up the list.
   }
}
```

e.g. Run each function on the list `head -> A -> B -> C -> D`.

A recursive call that does not reduce toward a base case is like an infinite loop. e.g. It is unknown whether this function terminates for `n > 0`. e.g. Try it with `n = 3`.

```
// What does this do for n > 0?
int bad(int n) {
   if (n == 1)
      return 1;
   if (n % 2 == 0)
      return bad(n / 2);
   return bad(3 * n + 1);
}
```

- Binary tree

  Vocabulary:

  - In a *binary tree*, each node has 0, 1, or 2 children, called *left* and *right*.
    e.g. draw `((5 - x) * y)`

- A *full* tree has levels $0, 1, 2, \ldots, n$ and 2^n leaves. e.g. draw `((3+4)*(1+2))`

- A *complete* tree is full or could be made full by adding leaves in the bottom level on the right. e.g. see `((5 - x) * y)` above

- array

  For a complete binary tree, there is a natural mapping into an array:

  * store the root at index `1`, leaving `0` empty
  * the parent of a node stored at `k` is stored at `k / 2`
  * the left child of a node stored at `k` is stored at `2 * k`
  * the right child of a node stored at `k` is stored at `2 * k + 1`

  e.g. Here is an array, `A`, storing `((5 - x) * y)`:

`arraySize`	8							
`treeSize`	5							
`i`	0	1	2	3	4	5	6	7
`A[i]`								

  (There are no gaps in the array for a complete binary tree.)

- linked via nodes of the form

```
template <typename T>
struct Tree {
    T data;
    Tree<T> *left;
    Tree<T> *right;
};
```

  e.g. Draw `((5 - x) * y)`:

root

How to insert and remove depends on the application.

To *traverse* a tree is to visit and process all its nodes. There are four standard tree traversals.

* Three are depth-first, using recursion (or a stack): *pre-order*, *in-order*, and *post-order*:

```
template <typename T>
void preorder(const Tree<T> *root) {
    if (root) {
        cout << root->data;
        preorder(root->left);
        preorder(root->right);
    }
}

template <typename T>
void inorder(const Tree<T> *root) {
    if (root) {
        inorder(root->left);
        cout << root->data;
        inorder(root->right);
    }
}

template <typename T>
void postorder(const Tree<T> *root) {
    if (root) {
        postorder(root->left);
        postorder(root->right);
        cout << root->data;
    }
}
```

e.g. Here is the preorder traversal of `((5 - x) * y)`:

```
pre(*)
    cout *
    pre(-)
        cout -
        pre(5)
            cout 5
            pre(nullptr)
            pre(nullptr)
        pre(x)
            cout x
            pre(nullptr)
            pre(nullptr)
    pre(y)
        cout y
        pre(nullptr)
        pre(nullptr)
```

output: ______________ (*prefix*)

e.g. inorder() gives *infix*, ______________, and postorder() gives *postfix*, ______________.

* One is breadth-first, using a queue.
  To traverse a tree in *level-order*:
  · Insert the root node in a queue.
  · Loop on removing a node, processing it, and inserting its children in the queue.

  e.g. Run level-order on `((5 - x) * y)`. (This is useful for printing a tree.)

  e.g. A level-order traversal of a complete tree gives the nodes in the order they are stored in an array.

Applications of trees include:

- parse trees in
  * mathematical expressions, e.g. see `((3+4)*(1+2))` above
  * compiling programming language text to machine instructions
  * syntax trees for natural language processing (NLP) tasks like speech recognition, machine translation, and optical character regognition
- character code tree for Huffman coding data compression
- decision / classification trees
- menus with (recursive) sub-menus
- a *binary search tree* (BST), in which each node has a key, and each node's key is larger than or equal to its left child's key and less than or equal to its right child's key, is useful for a key-value lookup table

• Priority queue

A priority queue allows inserting with priority and removing the highest-priority element.

Consider possibile implementations:

data structure	insert	remove	n of each
unordered list in array	$O(1)$	$O(n)$	$O(n^2)$
unordered linked list	$O(1)$	$O(n)$	$O(n^2)$
sorted list in array	$O(n)$	$O(1)$	$O(n^2)$
sorted linked list	$O(n)$	$O(1)$	$O(n^2)$
heap (below)	$O(\log_2(n))$	$O(\log_2(n))$	$O(n \cdot \log_2(n))$

Vocabulary:

- A binary tree is *heap-ordered* if its elements are non-increasing along each path from root to leaf.

– A *heap* is a complete binary tree, represented in an array, that is heap ordered.

Recall the array implementation of a complete binary tree, above. The array `A` of size `n + 1` contains a *heap* of size `n` if `A[i/2] >= A[i]` for every `i` in `[2, ..., n]`.

The operations on a heap are *insert* and *remove largest*. Each makes a small change, possibly violating the heap property, and then travels a vertical path to restore the heap.

– insert
  * add element `x` to the next open slot: `n = n + 1; a[n] = x`, which puts it at a leaf
  * *reheapify up* via `while (x > parent) { swap x with parent }`

  e.g. Insert S. (Hint: draw the heap.)

`arraySize`	8							
`n` (heap size)	5							
`i`	0	1	2	3	4	5	6	7
`A[i]`		T	G	Q	B	E		

  $O($______________$)$

– remove largest element
  * save the root from `A[1]` to return
  * move the last entry in the deepest level to the root via `A[1] = A[n]; n = n - 1`
  * *reheapify down* via `while (copied entry < either child) { swap with largest child }`
  * return saved former root

  e.g. Remove T.

`arraySize`	8							
`n` (heap size)	5							
`i`	0	1	2	3	4	5	6	7
`A[i]`		T	G	Q	B	E		

  $O($______________$)$

A priority queue (PQ) facilitates processing tasks in order of their priority. e.g.

– Repeatedly select the largest element.
  * Huffman coding uses a PQ repeatedly to find the two minimum-count trees.
  * A PQ is convenient and efficient for merging many sorted files into one.
  * Selection sort repeatedly does an $O(n)$ linear search to find the largest. Substitute an $O(log_2(n))$ PQ search to get heapsort (below).

– An operating system uses a PQ to schedule jobs on the CPU.

– A virtual memory system keeps as many "pages" of virtual memory (disk space) in actual memory as possible, removing the least-recently-used page when a new one is needed.

– Several graph algorithms, e.g. best-first search, use a PQ.

We'll discuss these topics soon:

- Graph
  - array
    - ...
    -
  - linked
    - ...
    -
  - graph traversals
    - depth first (via stack or recursion)
    - breadth first via queue
  - graph applications
    - ...
    -
- Search
  - linear search
  - binary search
  - hash table
- Sort
  - insertion
  - selection
  - quick
  - heap
  - merge