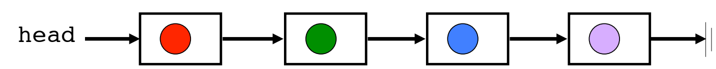
eecs 281 DATA STRUCTURES AND ALGORITHMS

Lecture 4: Linked List, Basic ADTs
(Review of Some 280 Material)
PA1 Walkthrough

Linked List Review

What is a linked list?



Each node points to the next node
The last node points to NULL

Linked Lists

```
struct Node {
  Node* next;
  int item;
  Node(){ next = NULL; }
};

class LinkedList{
  Node *head;

public:
  LinkedList();
  ~LinkedList();
  // other methods here
};
```

Linked Lists Methods

```
int getSize();

bool appendItem(int item);
bool appendNode(Node *n);

bool deleteItem(int item);
bool deleteNode(Node *n);
```

Why no `const` in `appendNode()`?

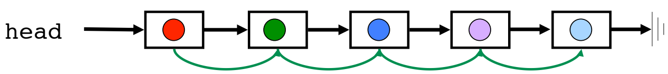
What if we wanted to store objects instead of just primitive types?

What other methods would be useful?

Counting Nodes in a Linked List

```
int LinkedList::getSize() {
    int i=0;
    Node *current = head;
    while (current){
      i++;
      current = current->next;
    }
    return i;
}
```

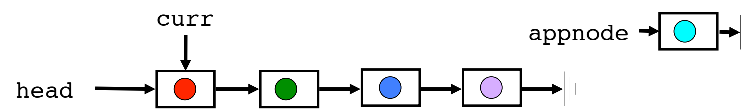
← traverses through the list



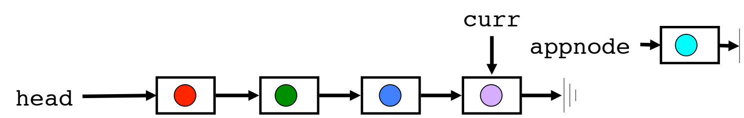
Visiting a series of elements is a traversal

Each step of a traversal is an iteration

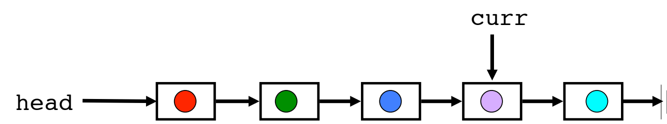
Linked List: Appending a Node



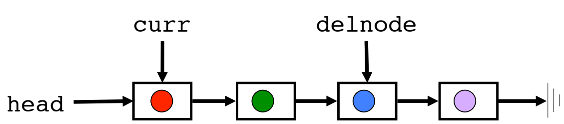
Search down list until `curr→next == NULL`



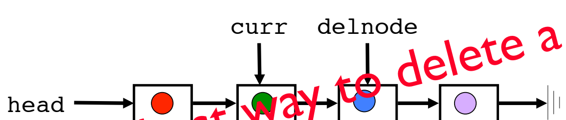
Set `curr→next = appnode; appnode→next = NULL`



Linked List: Delete Node

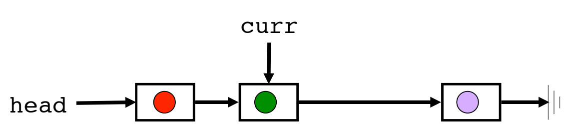


Search down list until `curr→next == delnode`

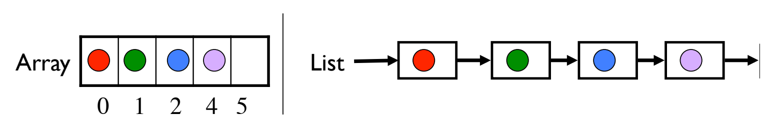


Not the best way to delete a node

Set `curr→next = delnode→next` and delete `delnode`



Arrays versus Linked Lists



	Arrays		Linked Lists	
Access	Random in $O(1)$ time	Worst Case	Random in $O(n)$ time	Worst Case
Insert Append	Inserts in $O(n)$ time Appends in $O(n)$ time		Inserts in $O(1)$ time Appends in $O(n)$ time	
Bookkeeping	Pointer to beginning Size or pointer to end of array		Pointer to first node Next node pointer in each node	
Memory	Free in $O(1)$ time Wastes memory if size is too large Requires reallocation if too small		Free in $O(n)$ time Allocates memory as needed Allocation/deallocation costly Next pointers wasteful	

Linked List Optimizations

```
class LinkedList {
   Node *head;
   int size;
public:
   LinkedList();
   ~LinkedList();
   int getSize() {
       return size;
   }
};
```

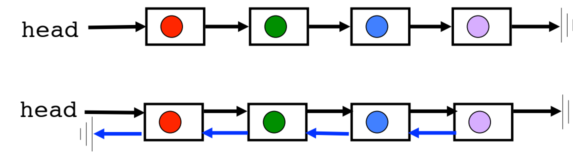
← incremented/decremented when nodes are inserted/deleted (int size;)

← returns stored size (return size;)

Doubly-linked List

Singly-linked

Doubly-linked



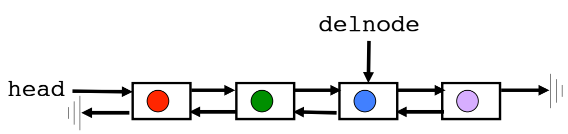
Each node points to next and previous nodes

First node's previous pointer and last node's next pointer point to NULL

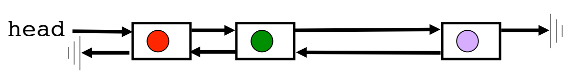
Which operations are faster with a doubly-linked list?

```
// Doubly-linked node
struct node{
   int item;
   Node* next;
   Node* prev;
   Node() { next = NULL;
            prev = NULL; }
};
```

Doubly-linked List: Delete Node




```
bool LinkedList::deleteNode(Node* delnode)
{
   delnode->prev->next = delnode->next;
   delnode->next->prev = delnode->prev;
   delete delnode;
   return true;
}
```



Complexity of Deleting a Node

Singly-linked list: $O(n)$
Doubly-linked list: $O(1)$

Why is deleting an element from a doubly-linked list so easy?

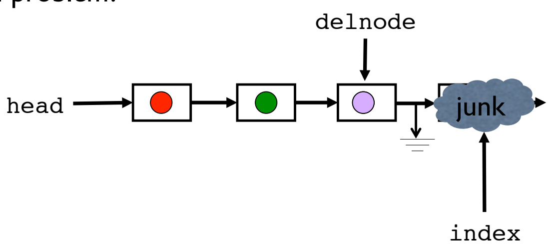
Why do we need $O(n)$ for a singly-linked list? Can we do better?

$O(1)$ Singly-linked Node Deletion

Idea 1:

- overwrite data in node to be deleted with next node's data
- delete next node
- assume data can be copied
  - some data such as references can not be copied
  - how to delete the last item on the list? Copy from head?
- potential problem?

Not recommended

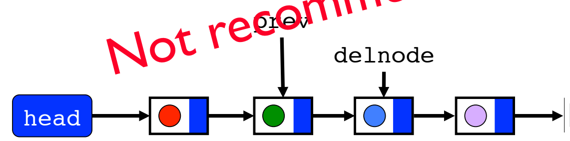


$O(1)$ Singly-linked Node Deletion

Idea 2:

- problem: we need access to the next pointer of the previous node
- how about we pass a pointer to the previous node instead? `deleteNode(Node *prev)`,

Not recommended

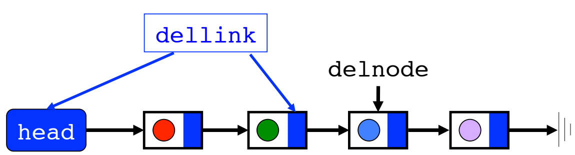


- new problem: how to delete the first node? `head` is a `Node *`, not a `Node`

$O(1)$ Singly-linked Node Deletion

Idea 3:

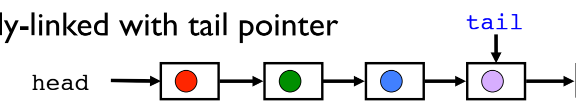
- problem: we need to access the next pointer of the previous node
- `typedef Node *Link;`
- `deleteNode()` called with a double pointer to the node to be deleted: `deleteNode(Link &dellink);`
- to delete a node in the linked-list (note the nifty use of reference args!): `deleteNode(prev→next);`
- to delete the first node: `deleteNode(head);`



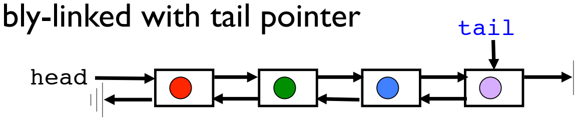
- use this method in your programming assignments

Linked Lists with Tail Pointers

Singly-linked with tail pointer



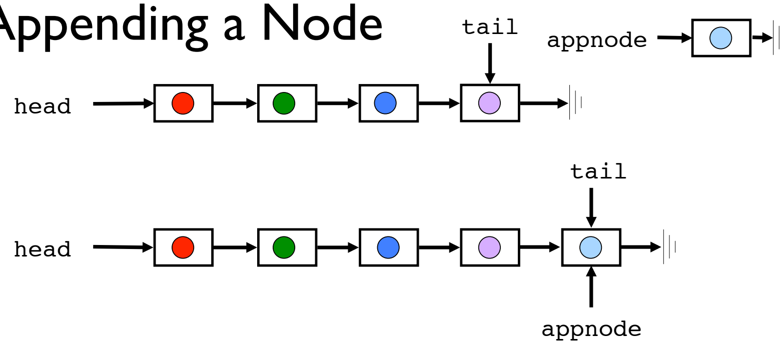
Doubly-linked with tail pointer




```
class LinkedList{
  Node *head;
  Node *tail;
public:
  // insert methods here
};
```

Which operations are faster with a tail pointer?

Singly-linked List with Tail Pointer: Appending a Node




```
void LinkedList::appendNode(Node* appnode){
  tail->next = appnode;
  tail = appnode;
  appnode->next = NULL;
}
```

Complexity of Appending a Node

Singly-linked List:

```
void LinkedList::addNode(Node* appnode){
  Node *current = head;
  while (current->next != NULL)
    current = current->next;
  current->next = appnode;
  appnode->next = NULL;
}
```

- `Node *current = head;` ← 1 step
- `while (current->next != NULL)` ← n steps
- `current->next = appnode;` ← 1 step

Total: $1 + n{*}1 + 1 = O(n)$

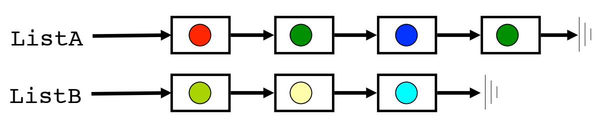
Singly-linked List with Tail Pointer:

```
void LinkedList::addNode(Node* appnode){
  tail->next = appnode;
  tail = appnode;
  appnode->next = NULL;
}
```

- `tail->next = appnode;` ← 1 step
- `tail = appnode;` ← 1 step

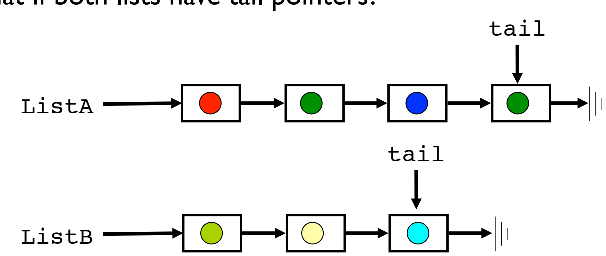
Total: $1 + 1 + 1 = O(1)$

Merging List



How long does it take to merge `ListA` and `ListB` into one list?

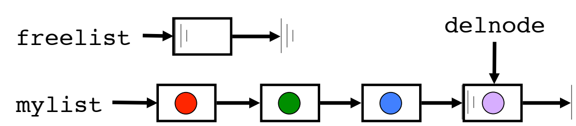
What if both lists have tail pointers?



Free List: Speeding Up (De)Allocation

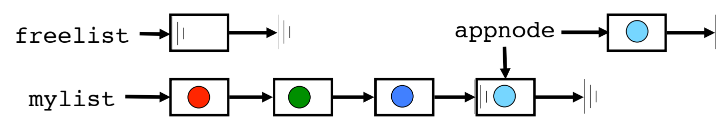
Prepend freed nodes to free list instead of de-allocating them

`mylist.deleteNode(delnode` ○ `)`



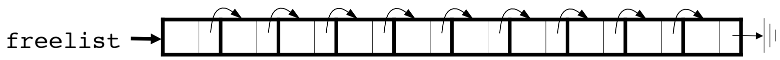
Next time you need a new node, take from free list first

`mylist.appendItem(`○`)` calls `mylist.appendNode(appnode` ○ `)`

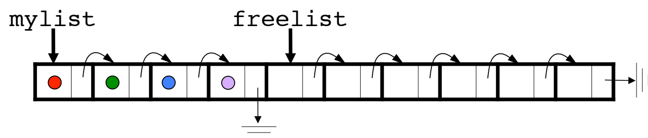


Free List: Implementation

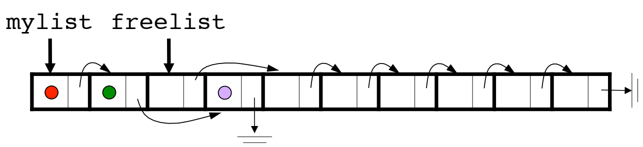
Pre-allocate a chunk of memory to create free list



with four nodes assigned to `mylist`:



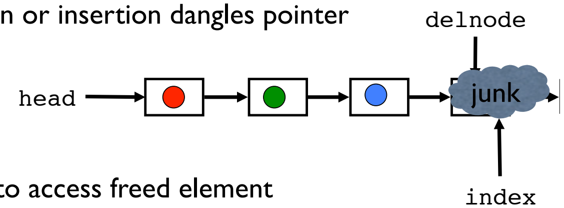
after `mylist.delNode(delnode` ● `)`:



Linked List: Common Bugs

Dangling pointer of one kind or another:

1. head and tail not initialized to NULL
2. free head without freeing each element
3. deletion or insertion dangles pointer



4. trying to access freed element

Consistency Checking

Arrays:

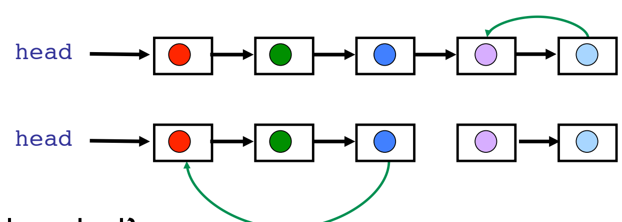
- does stored size match number of elements?
- check that `start+size < end`
  - `start`: pointer to start of array
  - `size`: stored size of the array
  - `end`: pointer to one slot past last element

Linked lists

- does stored size match number of elements?
- does the last node point to NULL?
- In a doubly-linked list, check that next/prev pointers are consistent `(p == p→next→prev)` and `(p == p→prev→next)`
- Is there a loop in the list?

Loop Detection

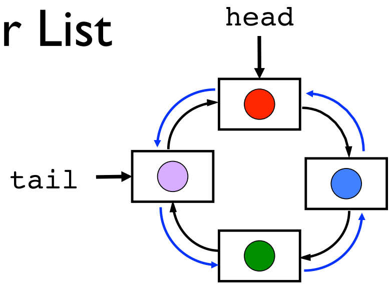
What is a "loop"?



Why is a loop bad?

How do you check for loop in a list?

Circular List

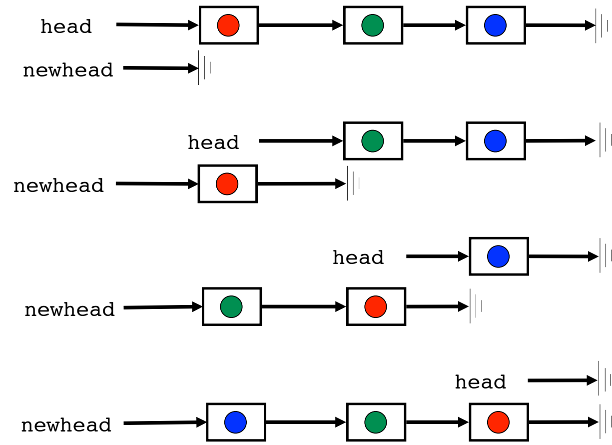


Last node points to first node

Can also be doubly-linked

Simplify coding: `prependNode()`, `appendNode()`, `insertNode()` can all use the same code

Reversing a Linked List



Now `head = newhead`

Complexity of Reversing a Linked List

How long does reversal take?

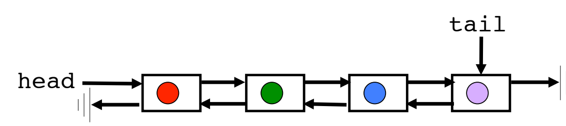
How much memory is needed?

Can reversal be made more time efficient?

Can reversal be made more space efficient?

- can we reverse with only $O(1)$ additional memory?

Reversing a Doubly-linked List



Can we reverse this in $O(1)$ memory?

Can we do the reversal in $O(1)$ time?

- what if we add a tail pointer?

Self-Study Questions

1. When would a linked list be preferred over an array?
2. How are linked lists and arrays similar?
3. What methods are faster with doubly-linked lists?
4. What methods are faster with a tail pointer?
5. What does it mean to check that a linked list is consistent?
6. How can you tell if a linked list is circular?

Abstract Data Type

What is Abstract Data Type (ADT)?

- a higher-level data representation (higher-level than arrays and linked-lists) that helps us conceptualize and manipulate a problem
- how ADTs are implemented is hidden from the users
- object-oriented ADTs come with pre-defined interfaces

Algorithmic Thinking: "Conceptualizes problems with digital representations and seeks algorithms that express or find solutions" – *P.J. Denning*

"(Almost) any problem in computer science can be solved by another layer of representation"

Vector ADT

What is a vector?
- a mathematical construct, a linear sequence of elements

How is a vector different from an array?
- boundary check is automatic
- resizing is automatic and invisible
- allows for operation on whole vector, e.g., add two vectors
- interfaces: `isempty()`, `getSize()`, `ithElement()`, `insert()`, `replace()`, `delete()`, etc.

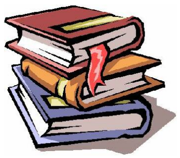
How to implement?

What is a Stack?

A "pile" of items where new objects are put on top of the pile and the top object is removed first (LIFO order)

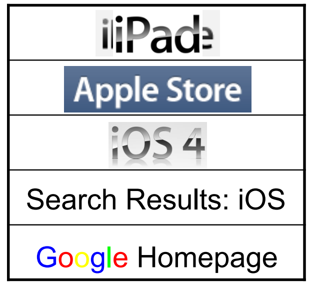
Method	Description
`push(object)`	add object to top of the stack
`pop()`	remove top element
`object& top()`	return a reference to top element
`size()`	number of elements in stack
`empty()`	checks if stack has no elements

Applications:
- Web browser's "back" feature
- Editor's "Undo" feature
- Function calls in C/C++



Stack Example: Web Browsing

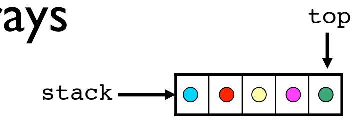
1. Open Browser to http://www.google.com
2. Search for "iOS"
3. Go to Apple's iOS4 page
4. Click on "Store"
5. Go to the "iPhone" page
6. Go back to "Store"
7. Go to the "iPad" page
8. Finished, close browser



url stack

Should we use arrays or linked lists to implement stacks?

Stacks Using Arrays

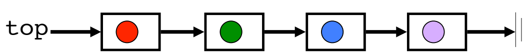


Keep a pointer (`top`) to the last element of array

Method	Implementation
`push(item)`	add new item to end of array and increment `top` allocate more space if necessary (requires copying)
`pop()`	decrement `top`
`Item& top()`	return reference to item in `top` node
`size()`	subtract `top` and `stack` pointers
`empty()`	are `top` and `stack` the same?

What is the asymptotic runtime of each method?

Stacks Using Linked Lists



Singly-linked list is sufficient

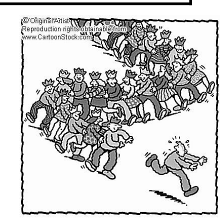
Method	Implementation
`push(item)`	prepend to the list
`pop()`	remove `top` of the list
`Item& top()`	return reference to item in `top` node
`size()`	Use `LinkedList::size()` method
`empty()`	Use `LinkedList::empty()` method

What is the asymptotic runtime of each method?
Is an array or linked list more efficient for stacks?

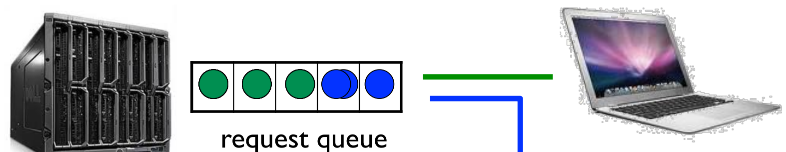
What is a Queue?

A "line" of items with FIFO access:
the first item inserted into the queue is the first one out

Method	Description
`enqueue(object)`	add element to tail of queue
`dequeue()`	remove element at head of queue
`Object& peek()`	return reference to element at head of the queue
`size()`	number of elements in queue, keep a count
`empty()`	checks if queue has no elements



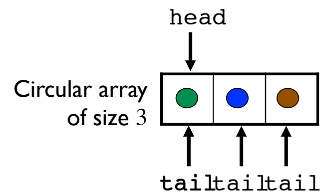
Queue Example: Request Queue of a Web Server

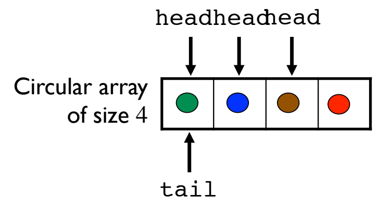


1. Computer A sends 2 requests
2. Computer B sends 3 requests
3. Server handles 3 requests
4. Computer A sends 1 request
5. Server handles 3 requests

Should we use arrays or linked lists to implement the server queue?

Queues Using Arrays: Enqueue and Dequeue

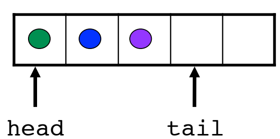




Event sequence

1. `tail == head` and count is 0
2. enqueue element
3. enqueue element
4. enqueue element, `tail++` modulo array len
5. allocate more (usually doubled) memory and enqueue element
6. dequeue element
7. dequeue element

Queues Using Arrays

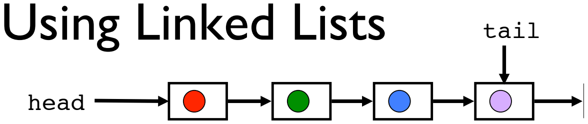


Use a circular array

Method	Implementation
`enqueue(item)`	increment `tail`, wrapping to front of array when end of allocated space is reached if `tail` becomes `head`, reallocate array and unroll
`dequeue()`	delete item at `head` and increment `head`, wrapping to front of array when end of allocated space is reached
`Item& peek()`	return reference to element at `head`
`size()`	`return count;`
`empty()`	`return count;`

What is the asymptotic runtime of each method?

Queues Using Linked Lists



Singly-linked list with tail pointer is sufficient

Method	Implementation
`enqueue(item)`	append to the list
`dequeue()`	delete `head` of the list
`Item& peek()`	return reference to item at `head` of list
`size()`	use `LinkedList::size()` method
`empty()`	use `LinkedList::empty()` method

What is the asymptotic runtime of each method?

Is an array or linked list more efficient for queues?

Deque: a Queue and Stack in One
(Double-ended Queue)

Not a proper English word, pronounced "deck"

Items can be inserted and removed from both ends of the data structure

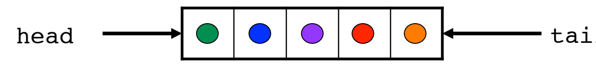
Six major methods
- `push()`, `dequeue()`, `peek_head()`
- `enqueue()`, `pop()`, `peek_tail()`

Minor methods
- `size()`, `empty()`

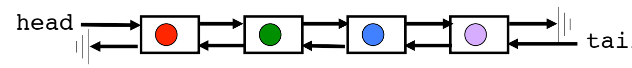
Cannot traverse elements

Deque Implementation

As circular array
- `head` and `tail` both get incremented/decremented



As doubly-linked list with tail pointer
- singly-linked doesn't support efficient removal
- other operations map directly to doubly-linked list operations



Which ADT to Use?

Freelist

The game Tetris

Strategy game:
- user can build multiple units
- but the barrack can only produce one unit at a time

Reverse Polish calculator:
- compute "$(5 + 9) * 2 + 6 * 5$" as "$5\ 9 + 2 * 6\ 5 * +$"
- advantage: no need to use parentheses to indicate precedence

Self-Study Questions

What is an ADT?

Define vector, stack, queue, deque

What is the best way to implement each of the ADT above?

Describe several applications where one ADT would be more appropriate than another

In choosing ADT for a given application, look for:
- the right trade-offs for runtime complexities
- memory overhead

Programming Assignment 1

Due date: Thu, 9/22, 10:00 pm

To be done individually (no group or team)

No STL (`iostream` and `string` are allowed, they are part of the C++ standard library, not part of STL)

Must not include external materials (e.g., open-source code downloads or code from previous terms from friends or found online)

To pass off the implementation of an algorithm as that of another is also considered cheating:

- e.g., insertion sort is *not* heap sort
- if you can not implement a required algorithm, you *must inform the teaching staff when turning in your assignment*

Problem Specification

Find a path from the given start position ('S') to the given target position ('T') in the game world

Game world consists of n levels of square maps

Levels are arranged left to right in sequential order, starting with `level 0`

`level 0`	`level 1`	`level 2`	`level 3`

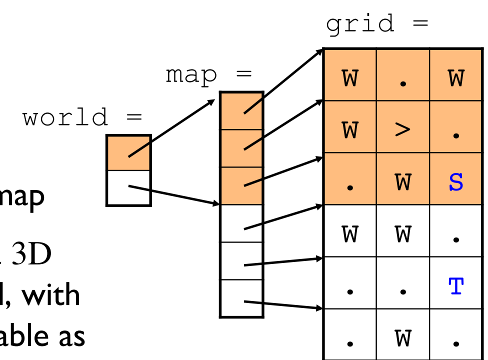
Each map consists of $m \times m$ number of tiles

Sample World Map

```
2
3
# sample input map file
# with two 3x3 level
W.W
W>.
.WS
WW.
..T
.W.
```

Actual has no color!

Map Representation



Each level has a map

You're creating a 3D map of the world, with each tile addressable as `world[level][row][col]`

'S' at `world[0][2][2]`
'T' at `world[1][1][2]`

Movements

You can move east, west, south, north, but not diagonally

You cannot move onto an impassable tile or off the map

The only movement you have at a portal is to exit at the destination level

- portals at location (x, y) on a level exits at the same location (x, y) on the destination level
- map coordinates start at $(0, 0)$ at the upper left (northwest) corner of the map

Path Finding

Start by inserting the start tile into a "navigation data structure", then (your implementation may vary):

1. visit next tile (search fail if no more next tile)
2. in a clockwise order starting from the tile north of this tile, examine each neighbor
3. if neighbor is not the target tile, is passable, and you have not previously visited this neighbor, insert it into the navigation data structure so that you can visit it later
4. loop (go back to step 1)

Your program must complete within 10 seconds of total CPU time

Path Finding

Two types of "navigator":

1. one stack based, built on a linked-list
2. one queue based, built on an array

Either one must be able to find the shortest path to the target

There is only one acceptable solution per algorithm for each input

Both algorithms must find a valid path (not necessarily the shortest) to the target if one exists

Path Output

Output a copy of the map with the path marked with the direction to take at each tile, starting from the start tile

The target tile should be marked with 'T'

Each level must be pre-tagged with "`#level k`"

```
2
3
#level 0
W.W
Wuw
.Wn
#level 1
WW.
.eT
.W.
```

Actual has no color!

Strip off comments, extra characters, and blank lines in the input file

If no path found, output original input map

dos2unix

Beware of DOS/Windows file format incompatibility:

- on Windows: line end is two characters ("\r\n")
- on Linux and Mac OS X: only '\n' is used
- many editors do not show the difference
- always run all files created on Windows through dos2unix on Linux before compiling and submitting
  - especially the testcases files!
  - by default, dos2unix overwrites input file with converted file

Remote Access

To login to remote Linux system

- from Linux/Mac OS X, use the ssh command
- from Windows: use putty (http://www.putty.nl/download.html)

To transfer files to/from remote Linux system

- from Linux/Mac OS X: use the scp command
- from Windows: use WinSCP (http://winscp.net/)

Beware of Linux/Windows file format incompatibilities

- always run all files (especially the testcases!) created on Windows through dos2unix on the Linux system before compiling and submitting

Files Organization

How would you organize your code into files?

Alternative 1: `path281.cpp` (NOT)

Alternative 2: `path281.cpp, navigator.h, stackNavi.h, stackNavi.cpp, queueNavi.h, queueNavi.cpp, stack.h, stack.cpp, queue.h, queue.cpp, linkedList.h linkedList.cpp, array.h, array.cpp` (NOT)

Alternative 3: `path281.cpp, navi.h, adts.h, tile.h`

Your choice would be different, but try not to split it up into too many files!

Time Requirements

How long does it take to do PA1?

Task	Lines of Code	% Total Time
design		6
world map	160	23
ADTs (unit test)	78 (+118)	21
navigator (unit test)	25 (+45)	6
path_finder()	133	16
path_printer()	138	13
clean up		15

PA1 Grading Criteria

Working, efficient solution (75%):

- code compiles
- code runs correctly, including two command line options: –q and –s
- implementation is efficient, e.g.,
  - no unnecessary copying
  - no loop invariant statement in loop

Test cases (20%)

Code is readable, well-documented (5%):

Code Readability

Negative space (or just plain space) increases code readability

Wrap around lines make code hard to read

Direct action is easier to follow than action description

Don't compute indices on the fly

Use short, mnemonic variable and function names

Code Reuse

Reuse code as much as possible

- reuse will reduce the amount of code to debug
- design for reuse
- put the code to be reused in a function, not cut-and-pasted

Coding Penalties (≤ 5 Points)

All code in one file	–2
Make clean not clean	–2
Junk file	–1

Per occurrence penalty:

Wrap around lines	–1
Cut-and-pasted code	–1
Meaningless variable/function names	–1
Use of literals	–1

Makefile

Make sure you don't have the following in your autograder output:

```
Warning: 'make clean' does not
remove all executable and object
files
```

See sample `Makefile` next slide

Makefile

```
CC = g++
CFLAGS = -g -Wall –pedantic

HDRS = navi.h adts.h tile.h
SRCS = path281.cpp
OBJS = $(patsubst %.cpp, %.o, $(SRCS))

path281: $(OBJS)
TAB $(CC) $(CFLAGS) -o $@ $(OBJS)

%.o: %.cpp $(HDRS)
TAB $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
TAB -rm -f *.o *~ *core* path281
```

Code Similarity

We use Moss to measure code similarity

Moss (a Measure Of Software Similarity) is an automatic system for determining the similarity of programs

Moss is a system for detecting software plagiarism

http://theory.stanford.edu/~aiken/moss/