Cpt S 122 – Data Structures

Abstract Data Types

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Abstract Data Types (ADTs)

- ADT is a set of objects together with a set of operations
  - Abstract
    - implementation of operations is not specified in ADT definition
    - E.g., List
  - Operations on a list: Insert, delete, search, sort
- C++ class are perfect for ADTs
- Can change ADT implementation details without breaking code that uses the ADT

Abstract Data Types (ADTs) (cont'd)

- Lists
- Stacks
- Queues

The List ADT

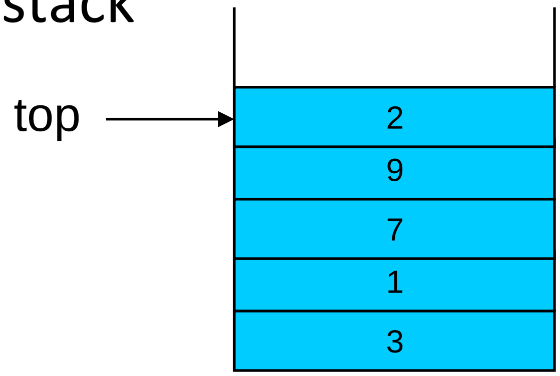
- List of size N: A_0, A_1, …, A_{N-1}
- Each element A_k has a unique position k in the list
- Elements can be arbitrarily complex
- Operations
  - insert(X,k)
  - remove(k)
  - find(X)
  - findKth(k)
  - printList()

The List ADT (cont'd)

- If the list is 34, 12, 52, 16 and 12
  - find(52) might return 2
  - insert(x, 2) might make the list into 34, 12, x, 52, 16, 12
  - remove(52) might turn the list into 34, 12, x, 16, 12

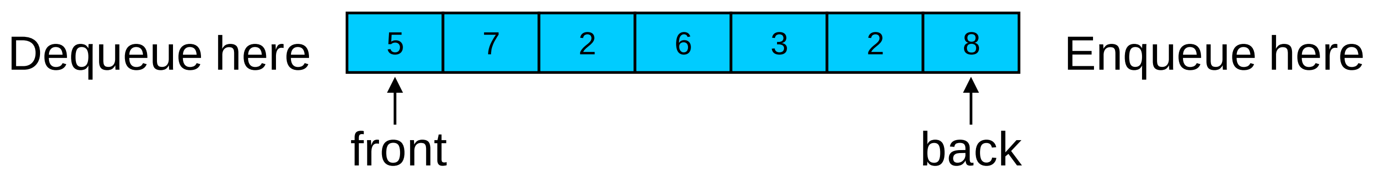
Stack ADT

- Stack is a list where insert and remove take place only at the "top"
- Operations
  - Push – inserts element on top of the stack
    - insertAtFront()
  - Pop – removes and returns element from top of the stack
    - removeAtFront()
  - Top – returns element at the top of the stack
- LIFO (Last In First Out)



Queue ADT

- Queue is a list where insert takes place at the back, but remove takes place at the front
- Operations
  - Enqueue – inserts element at the back of queue
    - insertAtBack()
  - Dequeue – removes and returns element from the front of queue
    - removeAtFront()
- FIFO (First In First Out)



Lists Using Arrays

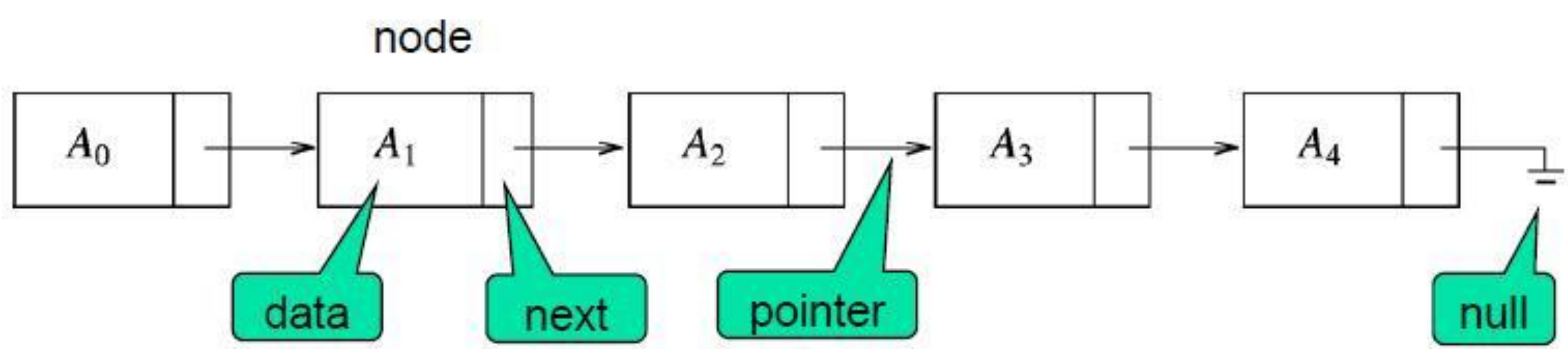
- Simple array vs. vector class in C++
  - Estimating maximum size
- Operations
  - insert(X, k) – O(N)
  - remove(k) – O(N)
  - find(X) – O(N)
  - findKth(k) – O(1)
  - printList() – O(N)

Lists Using Arrays (cont'd)

- Array implementation
  - printList() in linear time, findKth(k) operation takes constant time
  - insertion() and deletion() are potentially expensive based on where those occur
    - Front
    - Middle
    - End
  - Array is not a good option
    - Alternative is Linked List

Lists Using Linked List

- Elements are not stored in contiguous memory
  - Not necessarily adjacent in memory
- Nodes in list consist of data element and next pointer
  - Each node contains the *element* and a *link* to a node containing its successor
  - Link is called as *next* link
  - Last node's *next* link points to *NULL*

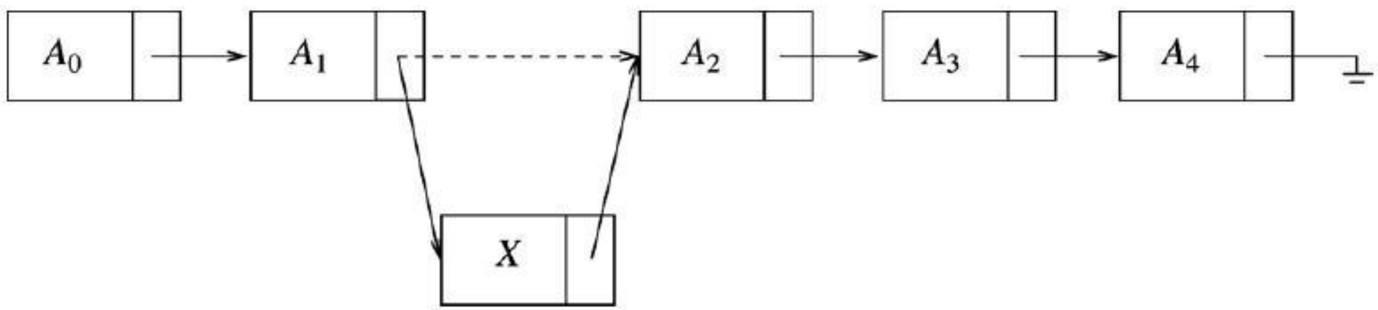


Lists Using Linked List (cont'd)

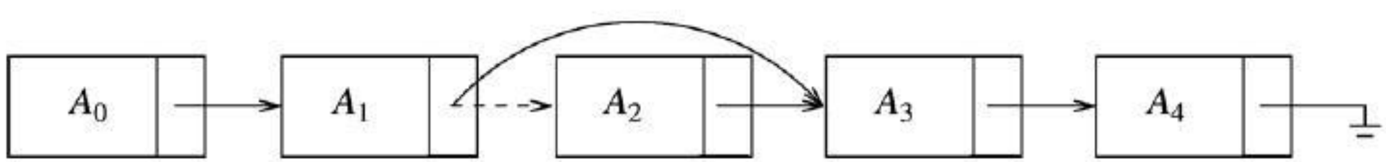
- Where a change is to be made if known, inserting and removing an item from a linked list does not require moving lots of items
  - Involves only a constant number of changes to the node links
- Special Cases:
  - adding to the front or removing the first item: constant time operation
  - adding at the end or removing the last item: constant time operation
    - Removing the last item is trickier
    - Find out the next-to-last item, change its *next* link to *NULL, a*nd then update the link that maintains the last node

Lists Using Linked List (cont'd)

- Operations
  - Insert(X, A) – O(1) (if we are already at the position to insert X and have another pointer pointing at previous node)
    - Only change of two pointers



  - Remove(A) – O(1) (if we are already pointing at A and have another pointer pointing at previous node)
    - Only change of one pointer

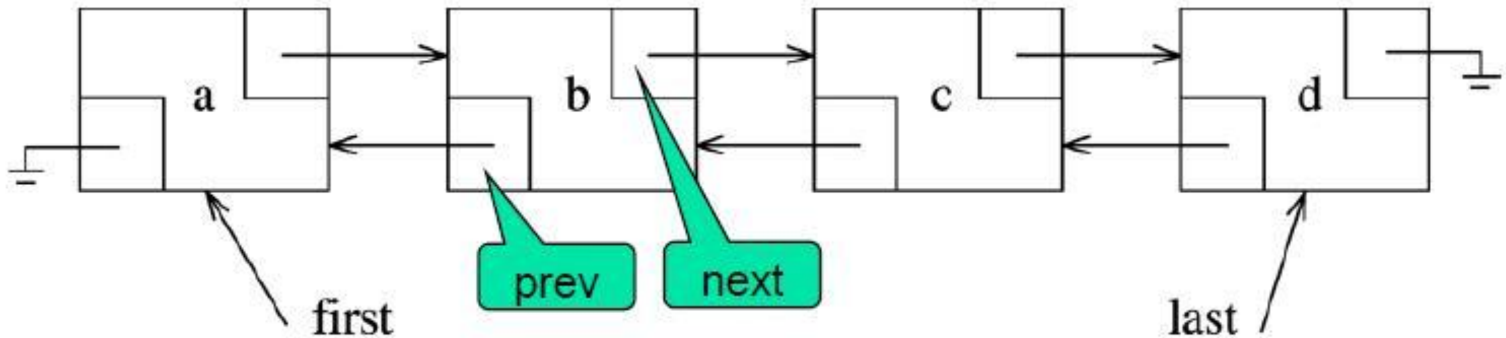


Lists Using Linked List (cont'd)

- Operations
  - find(X) – O(N)
  - findKth(k) – O(N)
  - printList – O(N)
- Start at the first node in the list and then traverse the list following the *next* links
  - find(X) – O(N)
  - printList – O(N)
- *findKth(k)* operation is no longer quite as efficient as an array implementation
  - It takes O(k) time and works by traversing down the list

Doubly-Linked List

- Singly-linked list
  - insert(X, A) and remove(X) require pointer to node just before X
- Doubly-linked list
  - Also keep pointer to previous node



Doubly-Linked List (cont'd)

- insert(X, A)

```
newA = new Node(A);
newA->prev = X->prev;
newA->next = X;
X->prev->next = newA;
X->prev = newA;
```

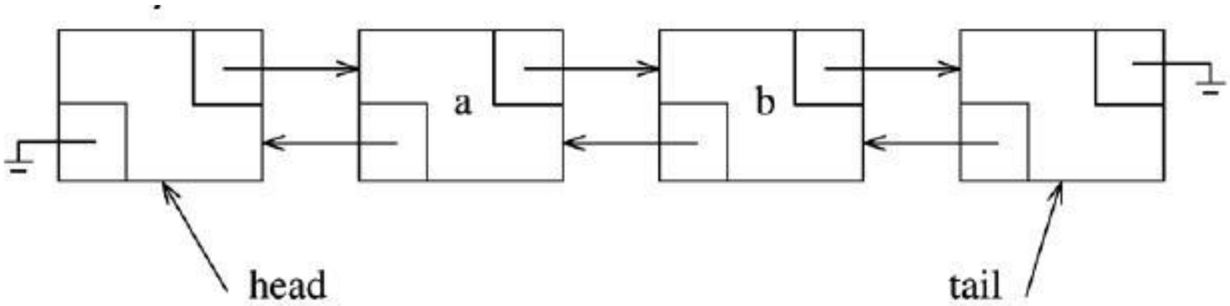
- remove(X)

```
X->prev->next = X->next;
X->next->prev = X->prev;
```

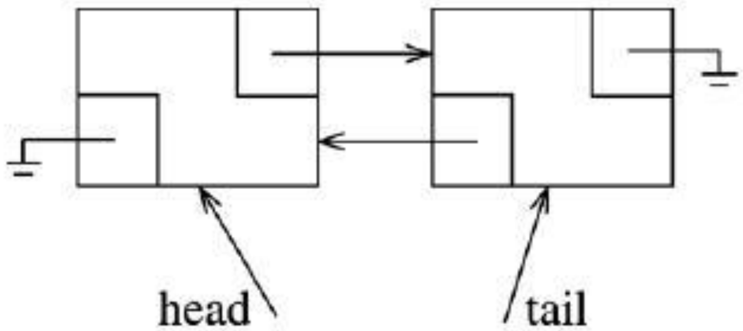
- Problems with operations at ends of list
  - Need to take care of special cases

Sentinel Nodes

- Dummy head and tail nodes to avoid special cases at ends of list
- Doubly-linked list with sentinel nodes



- Empty doubly-linked list with sentinel nodes



Lists Using STL

- Two popular implementation of the List ADT
  - The *vector* provides a growable array implementation of the List ADT
    - Advantage: it is indexable in constant time
    - Disadvantage: insertion and deletion are computationally expensive
  - The *list* provides a doubly linked list implementation of the List ADT
    - Advantage: insertion and deletion are cheap provided that the position of the changes are known
    - Disadvantage: list is not easily indexable
- Vector and list are class templates
  - Can be instantiated with different type of items

Lists Using STL (cont'd)

- **`vector<Object>`**
  - Array-based implementation
  - **`findKth`** – O(1)
  - **`insert`** and **`remove`** – O(N)
    - Unless change at end of vector
- **`list<Object>`**
  - Doubly-linked list with sentinel nodes
  - **`findKth`** – O(N)
  - **`insert`** and **`remove`** – O(1)
    - If position of change is known
- Both require O(N) for search

Common Container Methods

- `int size() const`
  - Return number of elements in container
- `void clear()`
  - Remove all elements from container
- `bool empty()`
  - Return true if container has no elements, otherwise return false

Vector and List Methods

- Both *vector* and *list* support adding and removing from the end of the list and accessing the front item in the list in constant time
- **`void push_back(const Object & x)`**
  - Add x to end of list
- **`void pop_back()`**
  - Remove object at end of list
- **`const Object & back() const`**
  - Return object at end of list
- **`const Object & front() const`**
  - Return object at front of list

List-Only Methods

- A doubly linked list allows an efficient changes at the front, but a *vector* does not, the following two methods are only available for *list*
- **`void push_front(const Object & x)`**
  - Add x to front of list
- **`void pop_front()`**
  - Remove object at front of list

Vector-Only Methods

- The *vector* has its own set of methods
  - Two methods allow efficient indexing
  - Other two methods to view and change internal capacity
- **`Object & operator[](int idx)`**
  - Return object at index idx in vector with no bounds-checking
- **`Object & at(int idx)`**
  - Return object at index idx in vector with bounds-checking
- **`int capacity() const`**
  - Return internal capacity of vector
- **`void reserve(int newCapacity)`**
  - Set new capacity for vector (avoid expansion)

C++ Standard Template Library (STL)

- Implementation of common data structures
  - Available in C++ library, known as Standard Template Library (STL)
  - List, stack, queue, …
  - Generally these data structures are called ***containers*** *or* ***collections***
- WWW resources
  - www.sgi.com/tech/stl
  - www.cppreference.com/cppstl.html