Computer Science 210: Data Structures

Linked lists
Arrays vs. Linked Lists

- We’ve seen arrays:
  - `int[] a = new int[10];`
  - `a` is a chunk of memory of size `10 x sizeof(int)`
  - `a` has a fixed size

```
```

- A linked list is fundamentally different way of storing collections
  - each element stores a reference to the element after it

```
  null
```
Arrays vs. Linked Lists

- **Arrays**
  - have a pre-determined fixed size
  - easy access to any element \( a[i] \) in constant time
  - no space overhead
    - Size = \( n \times \text{sizeof(element)} \)

- **Linked lists**
  - no fixed size; grow one element at a time
  - space overhead
    - each element must store an additional reference
    - Size = \( n \times \text{sizeof(element)} + n \times \text{sizeof(reference)} \)
  - no easy access to i-th element wrt the head of the list
    - need to hop through all previous elements
Linked-lists in Java

- Search for class Java LinkedList
- Has all expected methods and features
  - `add(int index, Object element)`
  - `add(Object o)`
  - `addAll(Collection c)`
  - `addAll(int index, Collection c)`
  - `addFirst(Object o)`
  - `addLast(Object o)`
  - `contains(Object o)`
  - `get(int index)`
  - `getFirst()`
  - `getLast()`
  - `indexOf(Object o)`
  - `lastIndexOf(Object o)`
  - `remove(int index)`
  - `remove(Object o)`
  - `removeFirst()`
  - `removeLast()`
  - `set(int index, Object element)`
  - `size()`
Implementing a linked list

- We want to implement a linked list class, much like Java’s LinkedList
- For simplicity, we can think of a linked list of integers
The Node class

We want to define the node in a list linked of integers.

/** Node of a singly linked list of integers */
public class Node {

...

}
The Node class

We want to define the node in a list linked of integers.

```java
/** Node of a singly linked list of integers */
public class Node {

    private int element;  //we assume elements are ints
    private Node next;

    ...  

}
```

self-referential definition
/** Node of a singly linked list of integers */
public class Node {
    private int element;  // we assume elements are ints
    private Node next;

    /** Creates a node with the given element and next node. */
    public Node(int s, Node n) {
        element = s;
        next = n;
    }

    /** Returns the element of this node. */
    public int getElement() { return element; }

    /** Returns the next node of this node. */
    public Node getNext() { return next; }

    // Modifier methods:
    /** Sets the element of this node. */
    public void setElement(int newElem) { element = newElem; }

    /** Sets the next node of this node. */
    public void setNext(Node newNext) { next = newNext; }
}
public class SLinkedList {
    protected Node head; // head node of the list
    protected long size; // number of nodes in the list

    public SLinkedList() {
        head = null;
        size = 0;
    }

    ...
A Single-Linked-List class

```java
/** Singly linked list.*/
public class SLinkedList {
    protected Node head;    // head node of the list
    protected long size;    // number of nodes in the list

    /** Default constructor that creates an empty list */
    public SLinkedList() {
        head = null;
        size = 0;
    }

    ...}

    • We’ll discuss the following methods
      • addFirst(Node n)
      • addAfter(Node n)
      • Node get(int i)
      • Node removeFirst()
      • addLast(Node n)
      • removeLast(Node n)
```
void addFirst(Node n) {
    n.setNext(head);
    head = n;
    size++;
}

• Notes
  • Special cases: works when head is null, i.e. list is empty
  • Efficiency: O(1) time
void insertAfter(Node v, Node n)

\[\text{v.setNext}(\text{n.getNext}());\]
\[\text{n.setNext}(\text{v.getN\text{ext}()});\]
\[\text{v.setNext}(\text{n});\]
\[\text{size}++;\]

- **Notes:**
  - Efficiency: \(O(1)\)
  - Special cases
    - does not work if \(v\) or \(n\) are null
    - null pointer exception
Get the i-th element

//return the i-th node

Node get(int i) {
    ...
}

Get the i-th element

//return the i-th node

Node get(int i) {
    if (i >= size) print error message and return null
    Node ptr = head;
    for (int k=0; k<i; k++)
        ptr = ptr.getNext();
    return ptr;
}

• Notes

• Special cases
  • does it work when list is empty?
• Efficiency: takes O(i) time
  • constant time per element traversed
  • unlike arrays, accessing i-th element is not constant time
Node removeFirst() {
    Node n = head;
    head = head.getNext();
    n.setNext(null);
    return n;
}

Notes:
- Special cases
  - does it work when list is empty?
    - Nope.
    - How to fix it?
- Efficiency: O(1)
void addLast(Node n) {
    insertAfter (get(size), n);
}

• Notes

• Special cases
  • does it work when list is empty?
    • Nope (first node in insertAfter is null).
    • How to fix it?

• Efficiency: takes O(size) time
Delete at tail

• **Remove at end: similar**
  
  • need to get to the last element from the head
  
  • $O(\text{size})$ time
Linked lists

- Single-linked lists support insertions and deletions at head in $\Theta(1)$ time.
- Insertions and deletion at the tail can be supported in $O(\text{size})$ time.

- addFirst: $O(1)$ time
- removeFirst: $O(1)$ time
- addLast: $O(\text{size})$ time
- removeLast: $O(\text{size})$ time

- Why? because we keep track of the head.
  - To access the tail in constant time, need to keep track of tail as well.
/** Singly linked list. */
public class SLinkedList {

    private Node head, tail; // head and tail nodes of the list
    private long size; // number of nodes in the list

    void SLinkedList() {
        head = tail = null;
        size = 0;
    }

    void addFirst(Node n) { ... }

    Node removeFirst() { ... }

    ....
}
void addLast(Node n) {

    //if list is empty the new element is head and tail
    if (tail == null) {
        n.setNext(null);
        head = tail = n;
    } else {
        //the list is not empty: link tail to n and n becomes the new tail
        tail.setNext(n);
        n.setNext(null);
        tail = n;
    }

    //increment size
    size++
}

• Special cases: list is empty
• Efficiency: Theta(1)
Remove at tail

• What we want: delete the last element and set the new tail
• Is that possible?
Remove at tail

- What we want: delete the last element and set the new tail
- Is that possible?

- Remove at tail
  - set the tail to the node BEFORE the tail
  - need the node before the tail: $O(\text{size})$

- To remove an element from a list you need the node BEFORE it as well

  ```
  remove(Node n) {
    //link n.before to n.next
  }
  ```

- To remove a node efficiently need to keep track of previous node
Doubly-linked lists

/** Node of a doubly linked list of integers */
public class DNode {
    protected int element;  // element stored by a node
    protected DNode next, prev;  // Pointers to next and previous nodes

    /** Constructor that creates a node with given fields */
    public DNode(int e, DNode p, DNode n) {
        element = e;
        prev = p;
        next = n;
    }

    /** Returns the element of this node */
    public int getElement() { return element; }

    /** Returns the previous node of this node */
    public DNode getPrev() { return prev; }

    /** Returns the next node of this node */
    public DNode getNext() { return next; }

    /** Sets the element of this node */
    public void setElement(Int newElem) { element = newElem; }

    /** Sets the previous node of this node */
    public void setPrev(DNode newPrev) { prev = newPrev; }

    /** Sets the next node of this node */
    public void setNext(DNode newNext) { next = newNext; }
}
Doubly-linked lists

/** Doubly linked list with nodes of type DNode */
public class DList {

    protected int size;  // number of elements
    protected DNode head, tail;

    void addFirst(Node n);
    void addLast(Node n);
    Node deleteFirst();
    Node deleteLast();
    void delete(Node n);
}

• Operations on doubly linked lists
  • addFirst():  O(1) time
  • addLast():   O(1) time
  • deleteFirst(): O(1) time
  • deleteLast(): O(1) time
  • delete():    O(1) time
  • get(i):      O(i) time
void addFirst(Node n) {
    n.setNext(head);
    n.setPrev(null);
    head.setPrev(n);
    head = n;
    size++;
}

Does this work?
void addFirst(Node n) {
    n.setNext(head);
    n.setprev(null);
    head.setPrev(n);
    head = n;
    size++;
}

• Special cases?
  • empty list: head is null; need to set tail too

• Efficiency ?
  • O(1)
void addLast(Node n) {
    tail.setNext(n);
    n.setPrev(tail);
    n.setNext(null);
    tail = n;
    size++;
}

Does this work?
void addLast(Node n) {
    tail.setNext(n);
    n.setPrev(tail);
    n.setNext(null);
    tail = n;
    size++;
}

• Special cases?
  • empty list: tail is null; need to set head too

• Efficiency: O(1)
Doubly-linked lists

• Class work: Sketch the following methods for doubly-linked lists, and analyze their efficiency.

• Node removeFirst()

• Node removeLast()

• void remove(Node n)

• Node search(int k)
Sentinels

- Sentinels for singly-linked list: keep a dummy head
  - an empty list is one node: the dummy head
- Sentinels for doubly-linked lists
  - dummy head and dummy tail
- Why? elegant. Unifies special cases when head or tail are null
public class DList {

    protected int size;  // number of elements
    protected DNode header, trailer; // sentinels

    /** Constructor that creates an empty list */
    public DList() {
        size = 0;
        header = new DNode(null, null, null);  // create header
        trailer = new DNode(null, header, null);  // create trailer
        // make header and trailer point to each other
        header.setNext(trailer);
    }

    • the empty list:
      • size = 0
DLLists with sentinels

```java
insertFirst(Node n) {
    n.setNext(dummyHead.getNext());
    dummyHead.getNext().setPrev(n);
    dummyHead.setNext(n);
    n.setPrev(dummyhead);
    size++;
}
```

- Special cases: none
  - works for empty list
Extensions

- Circular lists: make last node point to the first (instead of null)

```java
class CircularList {
    SNode head;
    int size;
}
```

- Let’s say we want to insert at head

```java
insertAtHead(Node n) {
    n.setNext(head.getNext());
    head.setNext(n);
}
```

- If head is null?

```java
if (head ==null) {
    n.setNext(n);
    head = n;
}
```