CS 261: Data Structures

Sorted Linked Lists
Complexity – Lists and Arrays

• Unordered linked have:
  – Fast Add operation: $O(1)$
  – Slow Search (Contains): $O(n)$

• Sorted Arrays have:
  – Slow Add operation: $O(n)$
  – Fast Search $O(\log n)$
Complexity – Lists and Arrays

• Unordered linked have:
  – Fast Add operation: $O(1)$
  – Slow Search (Contains): $O(n)$

• Sorted Arrays have:
  – Slow Add operation: $O(n)$
  – Fast Search $O(\log n)$

What about sorted lists?
Sorted Linked List

```
list
  sentinel
  val: 2
  next:
  val: 5
  next:
  val: 7
  next:
  val: 10
  next: null
```
Operations for Sorted Linked Lists

• Add:
  – find the correct location,
  – add the new element

• Contains:
  – find the correct location,
  – check if the element is in the list

• Remove:
  – find the correct location,
  – remove the element if found in the list
Sorted List Structure Definition

```c
struct list { 
    struct link * Sentinel;
    int size;
};
```
Find an Element in a Sorted List

```c
struct link * slideRightSortedList

(struct link *current, TYPE e) {
    assert(current);
    while (((current->next != 0) && LESS_THAN(current->next->value, e))
        current = current->next;
    return current; /* Returns the link RIGHT BEFORE */
}
```

- `e = 7`
Add Sorted List

```c
void addSortedList (struct list* lst, TYPE e) {
    assert(lst);
    struct link * current = slideRightSortedList(lst->Sentinel, e);
    ...}
```

```
e = 7
sentinel
val: 2
next:
val: 5
next: 
val: 10
next: null
```
void addSortedList (struct list* lst, TYPE e) {
    assert(lst);
    struct link * current = slideRightSortedList(lst->Sentinel, e);
    struct link * newLink = (struct link *) malloc(sizeof(struct link));
    assert (newLink != 0);
    newLink->value = e;
    ....
    newLink
}

e = 7
val: 2
next:

sentinel

val: 5
next:

current

val: 10
next: null
Add Sorted List

```c
void addSortedList (struct list* lst, TYPE e) {

    ... 

    newLink->next = current->next;
    /* For doubly linked lists */
    /* newLink->previous = current; */
    ...

}
```

---

**Diagram:**

- **sentinel**: `val: 2`\n  `next:`
- **current**: `val: 5`\n  `next:`
- **newLink**: `val: 10`\n  `next: null`
Add Sorted List

```c
void addSortedList (struct list* lst, TYPE e) {
    ...
    newLink->next = current->next;
    /* For doubly linked lists */
    /* newLink->previous = current; */
    /* current->next->previous = newLink; */
    current->next = newLink;
    lst->size++;
}
```

Diagram:
- **sentinel**
- **current**: val: 2
  - next: val: 5
    - next: val: 7
      - next: val: 10
        - next: null
void removeSortedList (struct list *lst, TYPE e) {
    struct link * temp; assert(lst);
    struct link * current =
        slideRightSortedList(lst->Sentinel, e);
    if ((current->next != 0) &&
        (EQ(current->next->value, e))){
        ....
    }
}
void removeSortedList (struct list *lst, TYPE e) {
    ...if ( ... ){
        temp = current->next
        current->next = current->next->next->next;
        /* For doubly linked lists */
        /* current->next->previous = current; */
        free(temp);
        lst->size--;
    }
}
Sorted Linked List

• What is complexity of:
  – search $O(??)$  
  – insertion $O(??)$  
  – removal $O(??)$

• Because we do not have a direct access to each link in the list
Problem with Sorted List

• What’s the use?
• Add, contains, remove $\Rightarrow$ $O(n)$
• No better than an unsorted list
• Major problem: sequential access
Sorted Linked List

• How to make a sorted linked list have faster operations?
Sorted Linked List

• Should I add more pointers?
  – E.g., add $\log n$ pointers
Adding more pointers…

• In theory this would work
• Would give us $O(\log n)$ search
• Hard to maintain insertion and removal