Lecture 7:
Introduction to the Lists
(Array-based linked lists)
Lists
Lists

• We used successive data structures up to now:
  
  • If \( a_{ij} \) in the memory location \( L_{ij} \), then \( a_{ij+1} \) is in \( L_{ij+c} \) (c: constant)
  • In a circular queue, if the \( i^{th} \) item is in \( L_i \), \((i+1)^{st}\) item is in \((L_i+c)\mod n\).
  • In a stack, if the top item is in \( L_T \), the below item is in \( L_{T-c} \).

Insertion and deletion: \( O(1) \)
Sequential Access
(ascending or descending)

Example 1:

- Alphabatically ordered lists:

  Ape  Butterfly  Cat  Dog  Mouse
  2  3

- Delete ‘Ape’, what happens?
- Delete ‘Cat’, what happens?
- Add ‘Bear’, what happens?
- Add ‘Chicken’, what happens?
**Sequential Access**
(ascending or descending)

**Example 1:**
- Delete ‘Ape’

<table>
<thead>
<tr>
<th></th>
<th>Ape</th>
<th>Butterfly</th>
<th>Cat</th>
<th>Dog</th>
<th>Mouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>Ape</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
</tr>
<tr>
<td>t1</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
<td></td>
</tr>
</tbody>
</table>
Sequential Access
(ascending or descending)

Example 1:

- Add ‘Ant’

<table>
<thead>
<tr>
<th>t0</th>
<th>Ape</th>
<th>Butterfly</th>
<th>Cat</th>
<th>Dog</th>
<th>Mouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Ape</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
</tr>
<tr>
<td>t2</td>
<td>Ape</td>
<td>Butterfly</td>
<td>Cat</td>
<td></td>
<td>Dog</td>
</tr>
<tr>
<td>t3</td>
<td>Ape</td>
<td>Butterfly</td>
<td></td>
<td>Cat</td>
<td>Dog</td>
</tr>
<tr>
<td>t4</td>
<td>Ape</td>
<td></td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
</tr>
<tr>
<td>t5</td>
<td>Ape</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
<td>Mouse</td>
</tr>
<tr>
<td>t6</td>
<td>Ant</td>
<td>Ape</td>
<td>Butterfly</td>
<td>Cat</td>
<td>Dog</td>
</tr>
</tbody>
</table>

- What if array is full?
Sequential Access  
(ascending or descending)

Example 2:

- The result of the multiplication of two polynomials
  - \((x^7 + 5x^4 - 3x^2 + 4)(3x^5 - 2x^3 + x^2 + 1)\)

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-2</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>-10</td>
<td>5</td>
<td>5</td>
<td>-9</td>
<td>6</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Powers are not ordered. So either we need to sort or shift in order to solve this problem.
Sorted items

- We want to keep the items sorted, and we want to avoid the sorting cost.
  - We may need to sort after each insertion of a new item.
  - Or we need to do shifting.

What is the solution?
Towards the Linked List

- Idea:
  - Each data item has a link (index) to the next data item
  - Each free item slot has a link to the next free item slot
  - We remember the locations of the first item and the first free slot

```
first = 0
free_ = 3
```
Initialisation

```c
#include <stdio.h>
#include <string.h>
#define MAX_LIST 6
#define TRUE 1
#define FALSE 0
#define NULL -1

typedef struct{
    char name[5];
    // other fields
    int link;
} item;

item linkedlist[MAX_LIST];
int first;
int free_;
void initialise()
{
    first = NULL;
    free_ = 0;
    for (int i = 0; i < MAX_LIST; i++)
        linkedlist[i].link = i + 1;
    linkedlist[MAX_LIST - 1].link = NULL;
}
get a free item slot

```c
int get_free_slot(int* slot) {
    if (free_ == NULL) // All slots are occupied
        return FALSE;
    else // Return the first free slot
        *slot = free_;
    return TRUE;
}
```
```c
int find_item(char name[]) {
    if (first == NULL) // list is empty
        return NULL;
    else { // iterate over items
        for (int next = first; next != NULL; next = linkedlist[next].link)
            if (strcmp(linkedlist[next].name, name) == 0)
                return next;
    }
    return NULL;
}
```
insert an item

first = 0
free_ = 3

insert("d")

first = 0
free_ = 4
insert an item

first = 0
free_ = 4

insert("bb")

first = 0
free_ = 5
insert an item

first = 0
free_ = 5

insert("_a")

first = 5
free_ = NULL
int insert_item(char name[]) {
    int free_slot;
    if (get_free_slot(&free_slot)) { // get_free_slot successful
        strcpy(linkedlist[free_slot].name, name);
        free_ = linkedlist[free_slot].link;
        int next = first;
        int prev = NULL;
        while (next != NULL && strcmp(linkedlist[next].name, name) < 0) {
            prev = next;
            next = linkedlist[next].link;
        }
        if (prev == NULL) { // Insert as the first item
            linkedlist[free_slot].link = first;
            first = free_slot;
        } else {
            linkedlist[free_slot].link = next;
            linkedlist[prev].link = free_slot;
        }
        printf("Item %s inserted. \n", name);
        return TRUE;
    } else { // No free slot exists
        printf("No free slot exists.\n");
        return FALSE;
    }
}
delete an item

first = 0
free_ = 5

delete("e")

NOT FOUND!
delete an item

first = 0
free_ = 5

delete("c")

first = 0
free_ = 2
delete an item

```
link: 1
```

```
0
```

```
a
```

```
b
```

```
d
```

```
bb
```

```
link: 4
```

```
link: 5
```

```
link: 3
```

```
link: 1
```

```
link: -1
```

```
link: 2
```

```
link: 3
```

```
link: -1
```

first = 0
free_ = 2

delete("a")

```
link: 2
```

```
link: 4
```

```
link: 5
```

```
link: -1
```

```
link: 3
```

```
link: -1
```

first = 1
free_ = 0
int delete_item(char name[]) {
    int next = first;
    int prev = NULL;
    while (next != NULL && strcmp(linkedlist[next].name, name) != 0) {
        prev = next;
        next = linkedlist[next].link;
    }

    if (prev == NULL)  // Deleting the first item
        first = linkedlist[first].link;
    else if (next != NULL) // Deleting normal item
        linkedlist[prev].link = linkedlist[next].link;
    else
        return FALSE;

    linkedlist[next].link = free_;
    free_ = next;
    return TRUE;
}
Example Code

- https://onlinegdb.com/rJk3gW76B
References

• Data Structures Notes, Mustafa Ege.