COMPUTER ENGINEERING 12
CLASS 14

Xiang Li
xli8@scu.edu

Project 4

So far, we have learned stack & queue (ADT)
We also finished linked list
To practice what we've learned in the lecture =>

We'll implement a stack and a queue through linked list!

Overview of Project 4

Week 1: A maze!

- stack

: A sorting!

- queue

It's an amazing sorting project!

You need to implement the deque using a circular, doubly-linked list with a sentinel or dummy node.

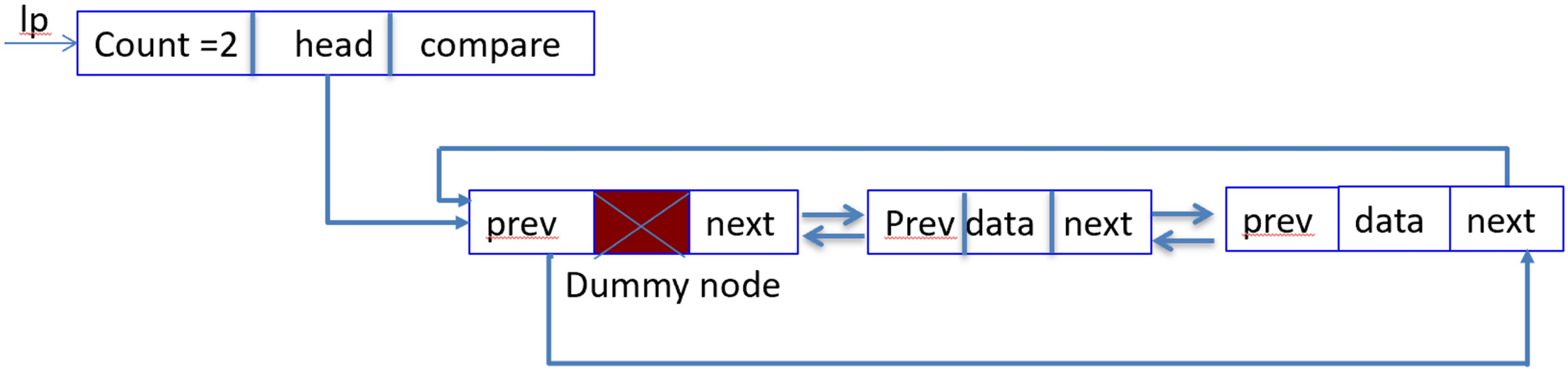
Week 2:

Since we've learned linked list, how about resolving collisions in your hash table by linked list (i.e. a chain).

Implement a generic SET ADT through hashing with chaining.

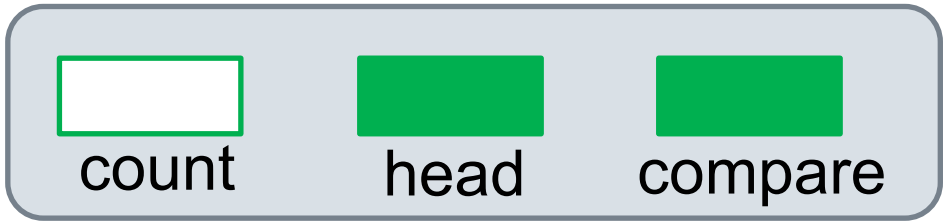
Implementation – A Deque (ADT)

- Week one
  - Implement a deque that can be used as either a queue or a stack
    - using doubly linked circular list with a dummy node



Structures

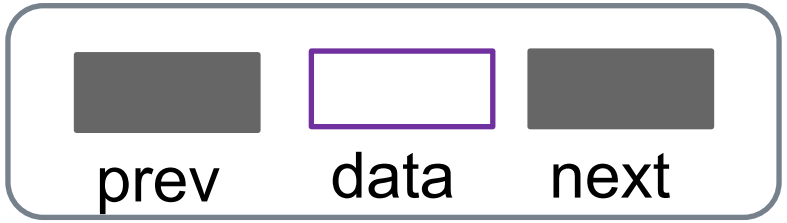
- List structure




```
struct list{
   int count;
   struct node *head;
   int (*compare)(); // function pointer
};

typedef struct list List; // list.h
```

- Data node structure

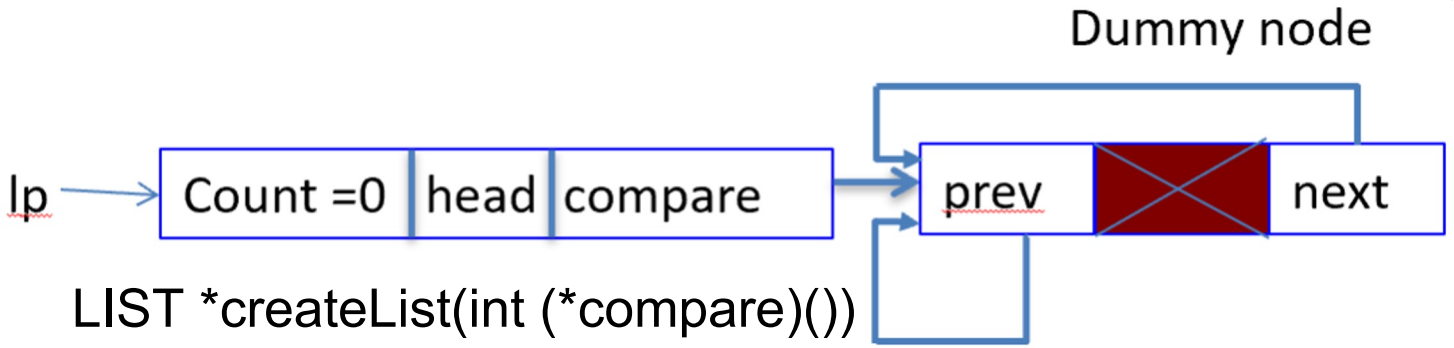



```
struct node{
   void *data;
   struct node *next;
   struct node *prev;
};

typedef struct node NODE; // list.h
```

Create a List with a Dummy Node

- Allocate a list and initialize it.



- Code:

```
LIST *createList(int (*compare)())
{
        struct list *lp;
        lp = malloc(sizeof(struct list));
        assert(lp != NULL);
        lp->count = 0;
        lp->compare = compare;
        lp->head = malloc(sizeof(struct node));
        assert(lp->head != NULL);
        lp->head->next = lp->head;
        lp->head->prev = lp->head;
        return lp;
}
```

There is no NULL pointer!

Recall - Destroy List with No Dummy Node

```
void destroyList(struct list *pList) {
    assert(pList != NULL);
    while (pList->head!=NULL){
        pDel = pList->head;
        pList->head = pDel->next;
        free(pDel);
     }
     free(pList);
}
```

Destroying a deque in this way will cause a problem!

You'll never have pList->head == NULL

Destroy a deque

```
void destroyList(LIST *lp)
{
        NODE *pDel, *pNext;

        assert(lp != NULL);
        pDel = lp->head;

        do {
             pNext = pDel ->next;
            free(pDel);
             pDel = pNext;
        } while (pDel!= lp->head);

        free(lp);
}
```

Destroy a deque

```
void destroyList(LIST *lp) {
    assert(lp != NULL);
    NODE *pPrev;
    NODE *pDel= lp->head->prev;
    while (pDel!=lp->head){
        pPrev = pDel->prev;
        free(pDel);
        pDel = pPrev;
    }
     free(pDel);
     free(lp);
}
```

Deleting the list from the last element.

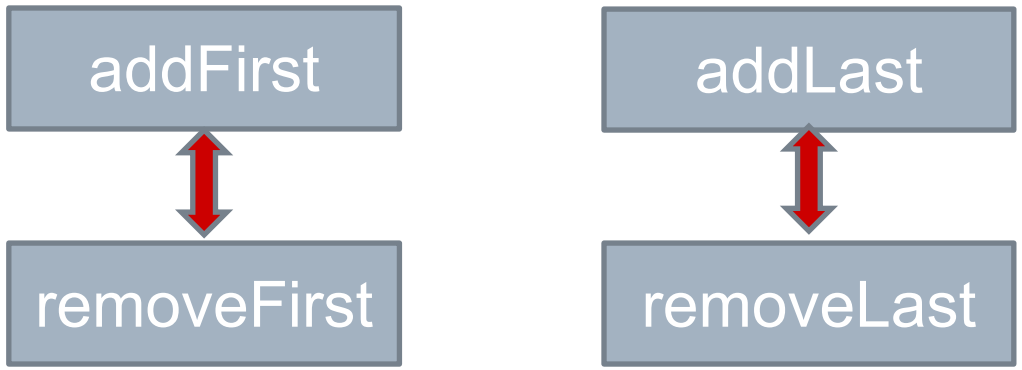
Other Functions

- extern LIST *createList(int (*compare)());
- extern void destroyList(LIST *lp);
- extern int numItems(LIST *lp);
- extern void addFirst(LIST *lp, void *item);
- extern void addLast(LIST *lp, void *item);
- extern void *removeFirst(LIST *lp);
- extern void *removeLast(LIST *lp);
- extern void *getFirst(LIST *lp);
- extern void *getLast(LIST *lp);
- extern void removeItem(LIST *lp, void *item);
- extern void *findItem(LIST *lp, void *item);
- extern void *getItems(LIST *lp);

Why do we need the same operation at both ends?

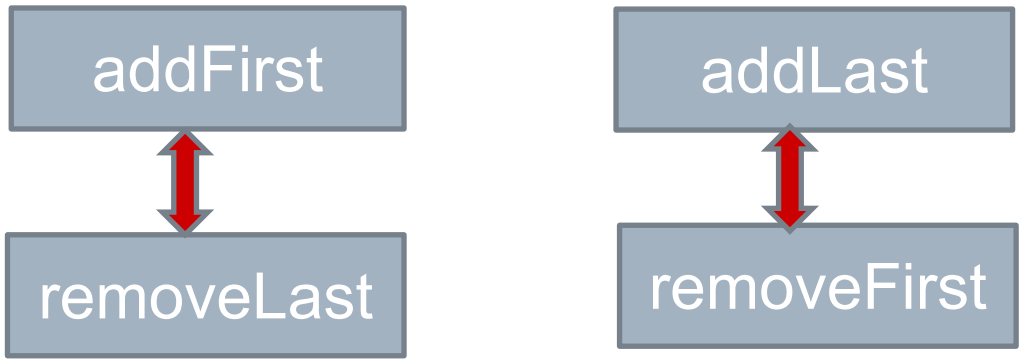
To Have a Stack

The outsider program (i.e. maze.c) needs to call a pair of functions for insertion and deletion



To Have a Queue

The outsider program (i.e. radix.c) needs to call a pair of functions for insertion and deletion

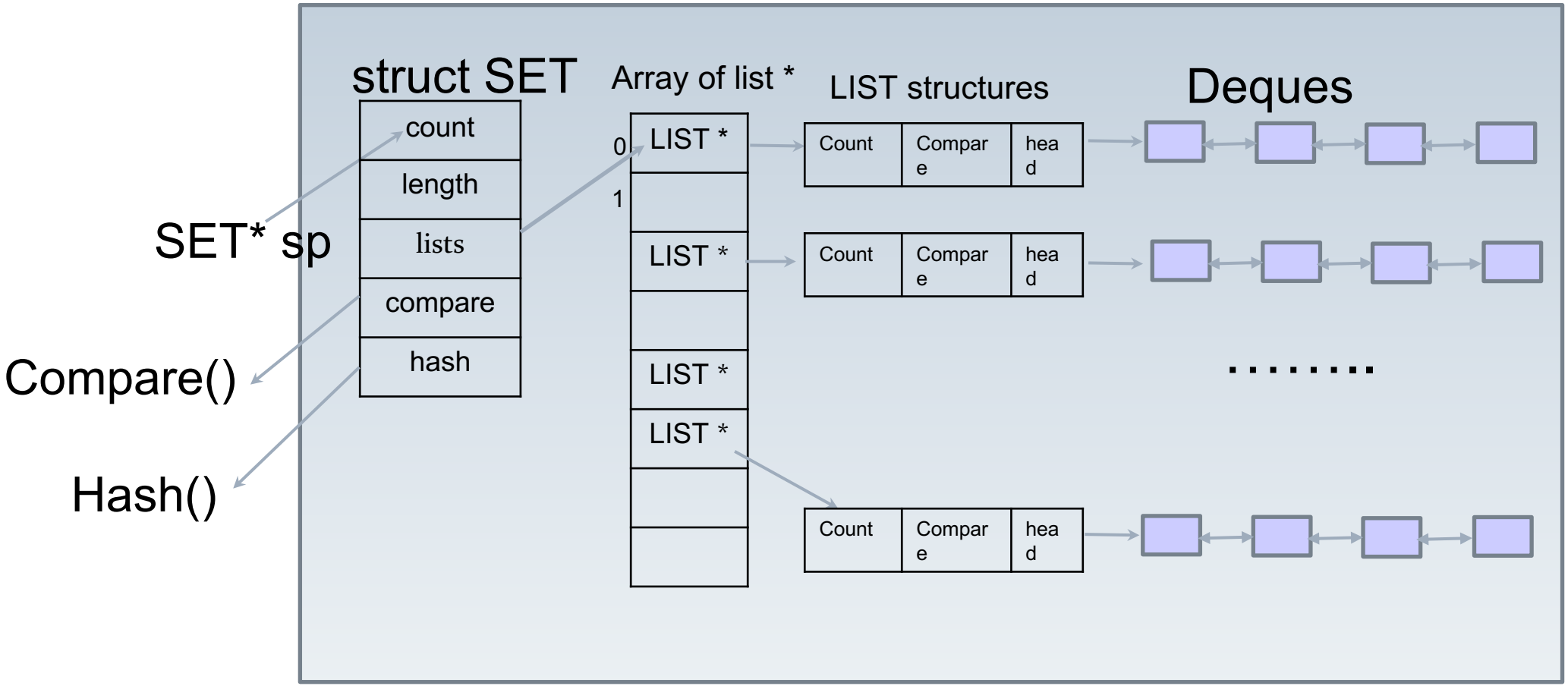


Week 2 – A Generic SET ADT

Since we've learned linked list, how about resolving collisions by using hash table with chaining?

Hint: the chain is the deque you have implemented in week 1.

The Data Structures



createSet

```
SET *createSet(int maxElts, int (*compare)(), unsigned (*hash)()){
        int i;
        SET *sp;
        assert(compare != NULL && hash != NULL);
        sp = malloc(sizeof(SET));
        assert(sp != NULL);
        //initialize length; count; compare and hash;
        sp->length =? ;
        sp->compare = compare;
        sp->hash = hash;
        sp->count = 0;
        //initialize array of lists: allocating memory for it.
        // for each element in the lists array (e.g. sp->lists[i]), create a list,
        //and let sp->lists[i] point to it.
        return sp;
}
```