BLM5106- Advanced Algorithm Analysis and Design

H. İrem Türkmen

Introduction to the Design and Analysis of Algorithms, Anany Levitin
http://ocw.mit.edu, Design and Analysis of Algorithms
http://web.stanford.edu/class/archive/cs/cs161/cs161.1176/, Design and Analysis of Algorithms
https://ceng.metu.edu.tr/, Data Structures

Substitution by Mathematical Induction

- Master Teorem does not work in this case and recursion trees can get pretty messy here, since we have a recurrence relation that doesn't nicely break up our big problem into sub-problems of the same size.
- Instead, we will try to:
- Make a guess
- Check using an inductive argument
- This is called the substitution method : Assume your guess is true for 1,2,..,n-1, using this assumption you have to proof that your guess is also true for n

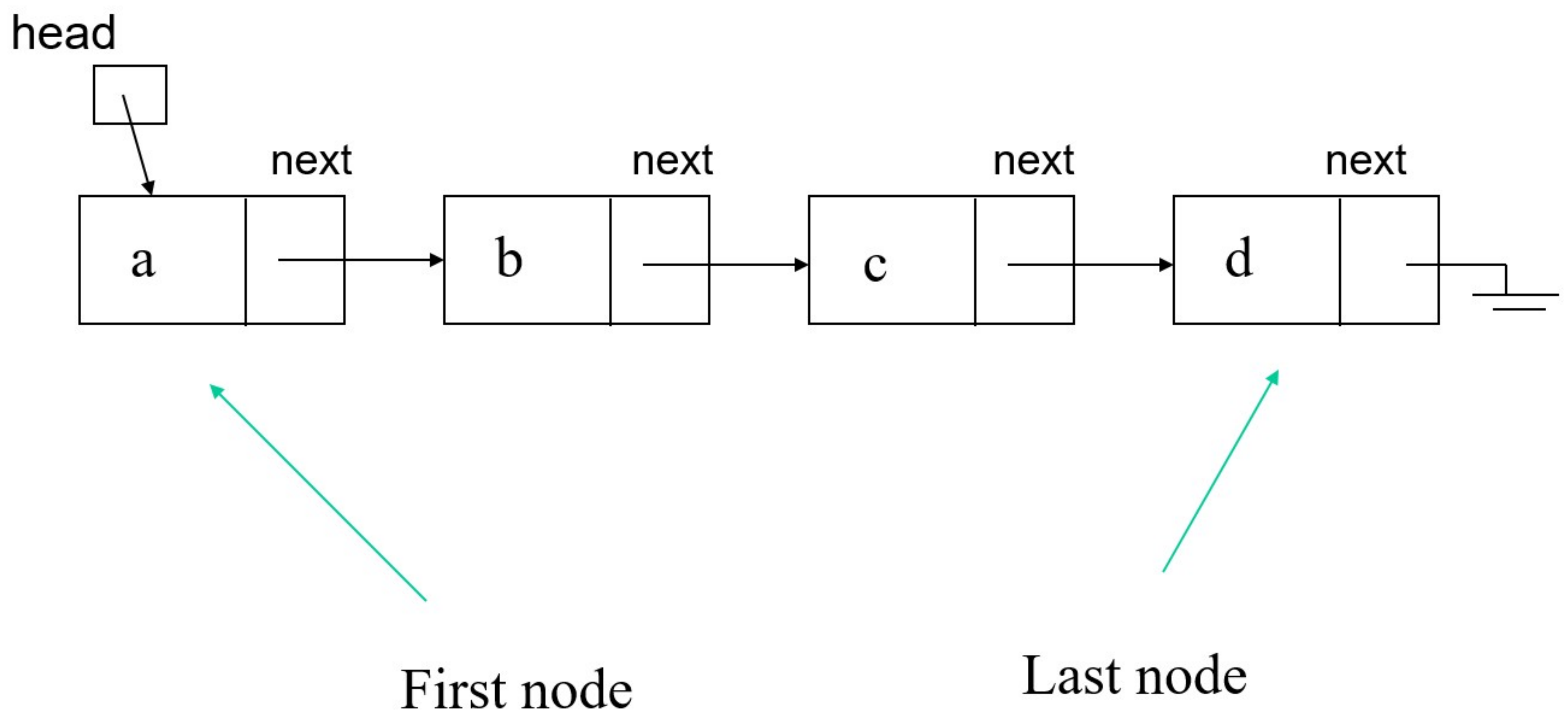
Substitution by Mathematical Induction

- Lets see some examples
- How to analyze devide and conquer median finding by mathematical induction?

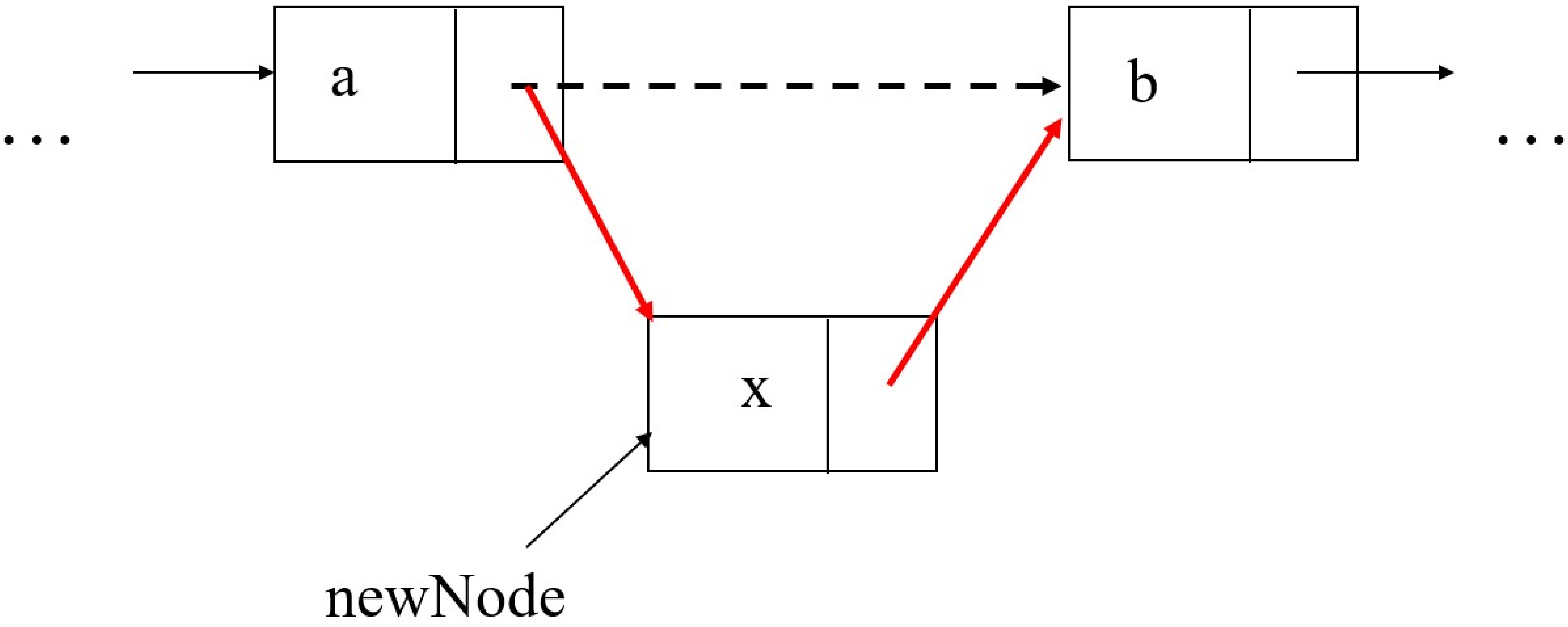
Basic Data Structures: Linked Lists

- Lets remember linked lists..
- Linked lists are used to store a collection of information (like arrays)
- A linked list is made of nodes that are pointing to each other
- We only know the address of the first node (head)
- Other nodes are reached by following the "next" pointers
- The last node points to NULL

Single Linked Lists



Insert a node

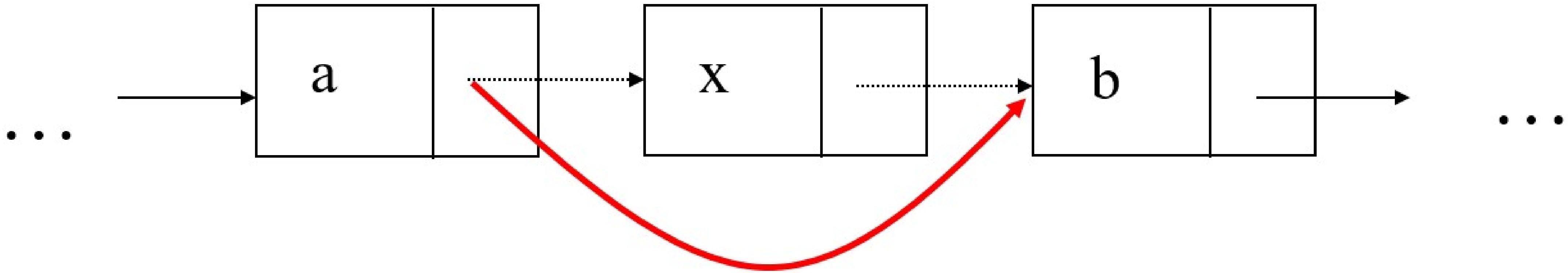


```c
struct node{
    int val;
    struct node *next;
};

// listenin sonuna node ekler
void push(struct node *head, int val) {
    struct node* current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = malloc(sizeof(struct node));
    current->next->val = val;
    current->next->next = NULL;
}
```

```c
// listenin basina node ekler
struct node* pushhead (struct node *head, int val) {
    struct node* newN;
    newN=malloc(sizeof(struct node));
    newN->next =head;
    newN->val =val;
    return newN;
}
```

```c
// elemanları listeler
void list(struct node *head)
{
    struct node* current = head;
    printf("liste elemanlari:\n");
    while (current->next != NULL) {
        printf("-%d-",current->val);
        current = current->next;
    }
    printf("-%d-\n",current->val);
}
```

Delete a node



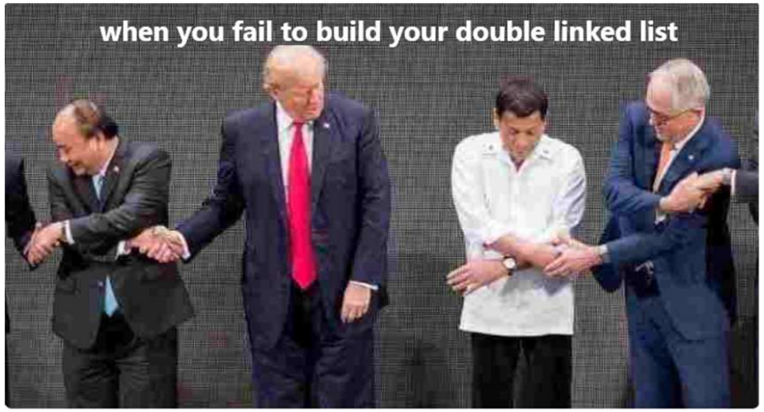
```c
void deleteNode(struct node *head, int val)
{
    struct node* current = head, *before=NULL;

    while ((current->val != val)&&(current->next != NULL))
    {
        before=current;
        current = current->next;
    }

    if (current->val != val)
        printf("silinmek istenen eleman listede yok\n");
    else
    {
        before->next=current->next;
        free (current);
    }

}
```

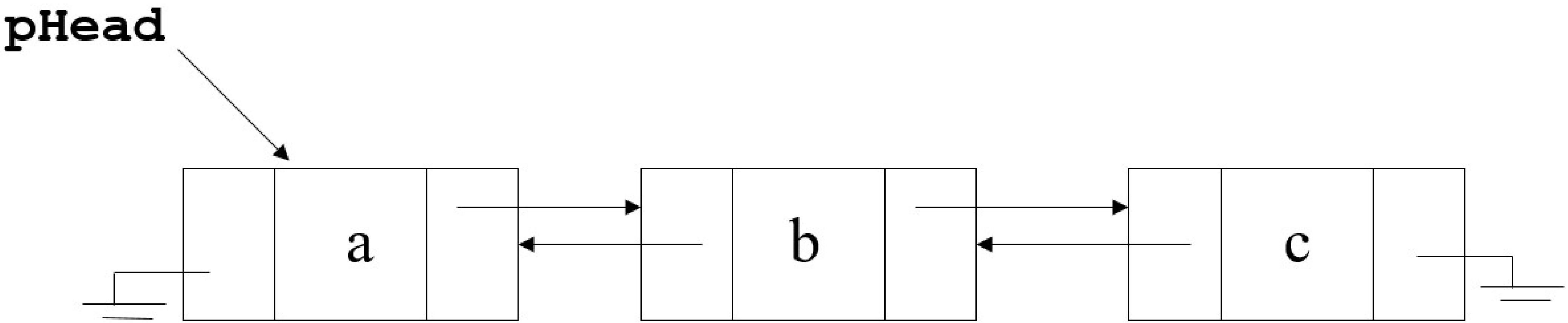
Single Linked Lists

- Singly linked lists in data structures can grow and shrink during runtime as needed without a predefined size.
- They only allocate memory for nodes that are actually in use, reducing memory waste compared to pre-allocated data structures like arrays.
- Adding or removing nodes doesn't require shifting elements (O(1) for linked lists), which can be a costly operation in arrays (O(N) for arrays). This is particularly beneficial at the beginning of the list.
- Used in implementing Stacks and Queues
- In systems programming, singly linked lists are used to track free memory blocks in dynamic memory allocation

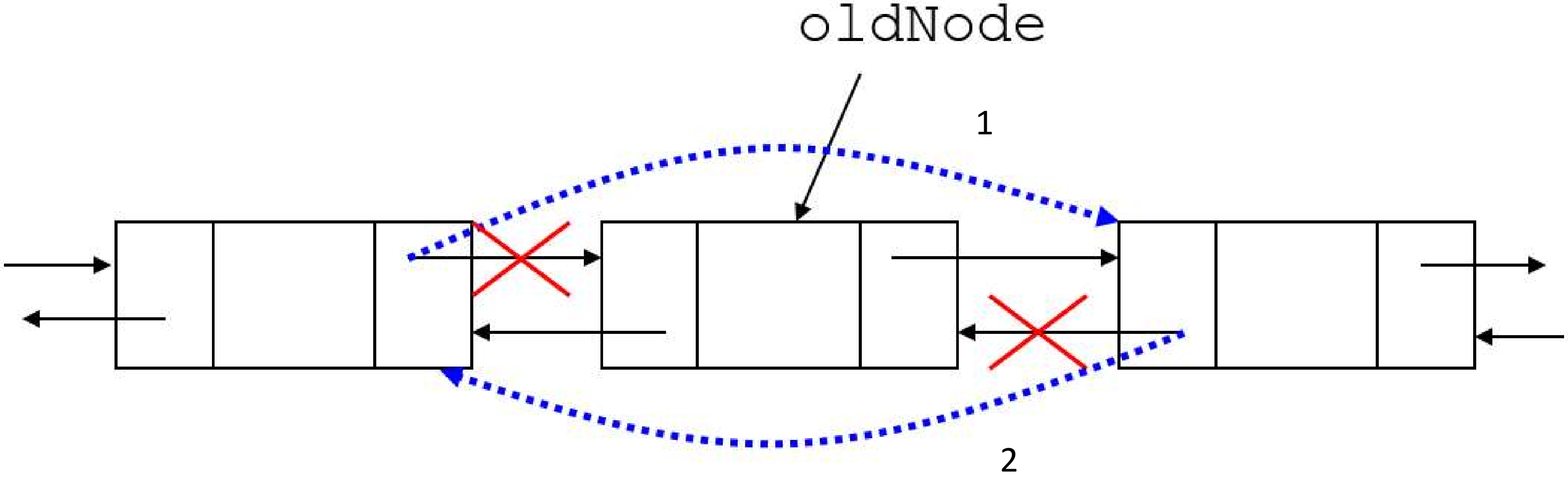
Double Linked List

- Convenient to traverse the list backwards (exp printing the contents of the list in backward order)
- Increase in space requirements due to storing two pointers instead of one



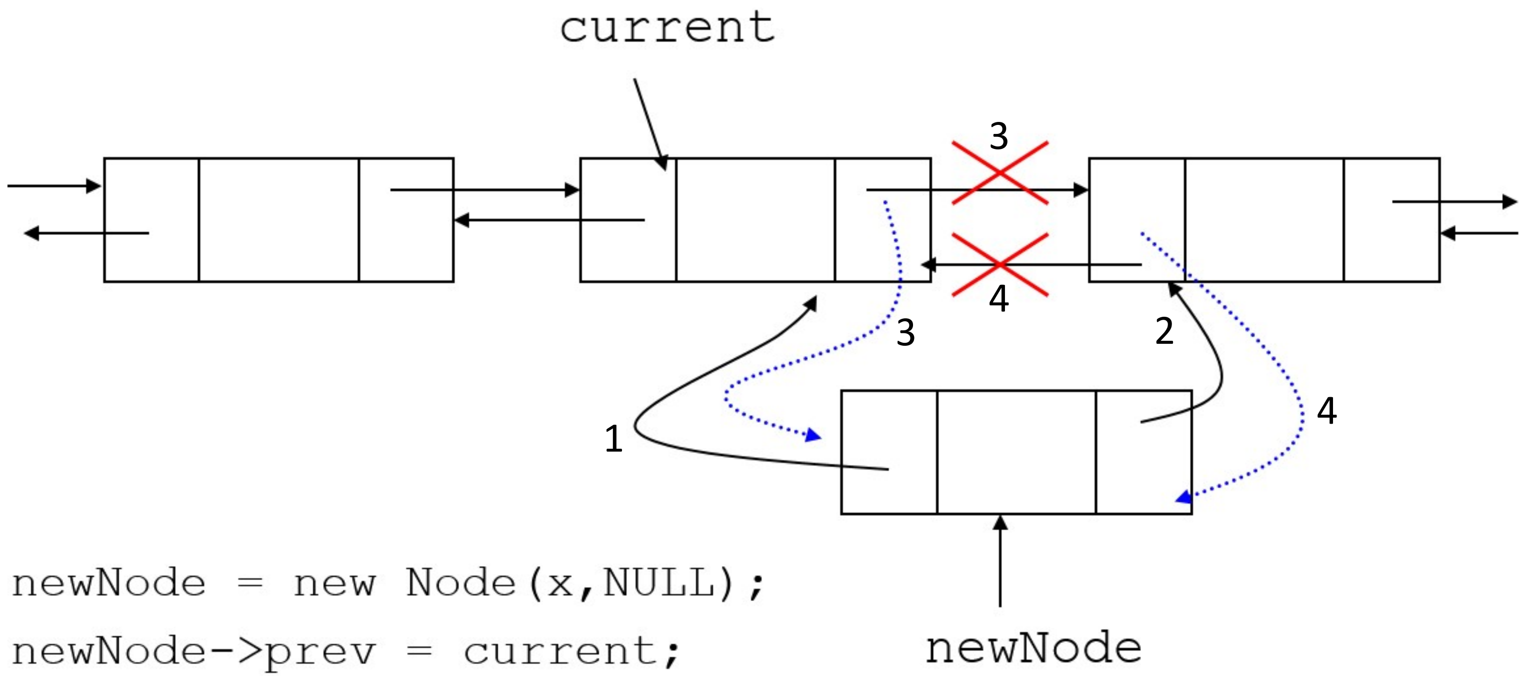


Delete a Node of Double Linked List




```
1  oldNode->prev->next = oldNode->next;
2  oldNode->next->prev = oldNode->prev;
   delete oldNode;
```

Insert a Node to Double Linked Lİst



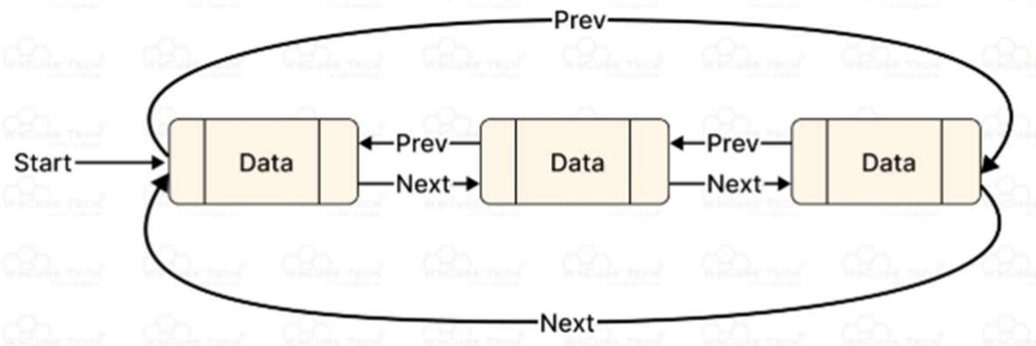

```
  newNode = new Node(x,NULL);
1 newNode->prev = current;
2 newNode->next = current->next;
3 newNode->prev->next = newNode;
4 newNode->next->prev = newNode;
```

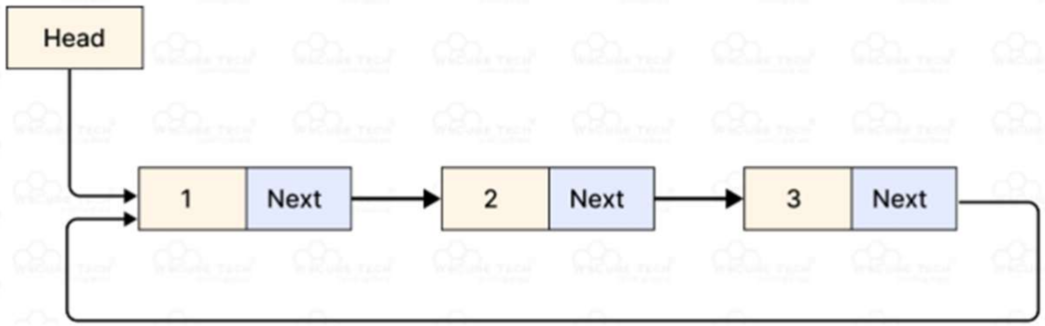
Double Linked List

- Reversing the doubly linked list is very easy.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node (O(N)) to be deleted but in the doubly linked list, it only required the pointer which is to be deleted (O(1))
- Implementing graph algorithms
- Applications to implement undo and redo functionality
- Navigation systems where front and back navigation is required
- Least Recently Used (LRU) cache implementation..

Circular Linked List

- Divide & Conquer Convex Hull
- Repeat and shuffle functions in music and video players,
- In games with multiple players, to manage player turns in a loop,
- In token ring networks (where each computer on a network is given a chance to transmit data in a pre-defined, circular order)





A very efficient way to implement dictionaries: HASHING

What is a dictionary?

- Dictionary is an abstract data type, a set with the operations of searching (lookup), insertion, and deletion defined on its elements.
- The elements of this set can be of an arbitrary nature: numbers, characters of some alphabet, character strings, and so on.
- Student records in a school, citizen records in a governmental office, book records in a library..

Dictionary

- Typically, records comprise several fields, each responsible for keeping a particular type of information about an entity the record represents.
- For example, a student record may contain fields for the student's ID, name, date of birth, sex, home address, major, and so on.
- Among record fields there is usually at least one called a ***key*** that is used for identifying entities represented by the records (e.g., the student's ID).
- We assume that we have to implement a dictionary of n records with keys $K_1, K_2, \ldots, K_n$.

Hash Table and Hash Function

- ***Hashing*** is based on the idea of distributing keys among a one-dimensional array $H[0..m - 1]$ called a ***hash table***.
- The distribution is done by computing, for each of the keys, the value of some predefined function h called the ***hash function***.
- This function assigns an integer between 0 and $m - 1$, called the ***hash address***, to a key.

Hash Table

- Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE.
- For example, in complilers Symbol tables are often implemented as hash tables because a compiler must be able to store and retrieve information about symbols very quickly
- Although searching for an element in a hash table can take as long as searching for an element in a linked list (O(n) time in the worst case), under reasonable assumptions, the average time to search for an element in a hash table is O(1).

Hash Table

- When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored.

Direct – address Tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U={0,1,2,..,m-1}, where m is not too large.
- We shall assume that no two elements have the same key.

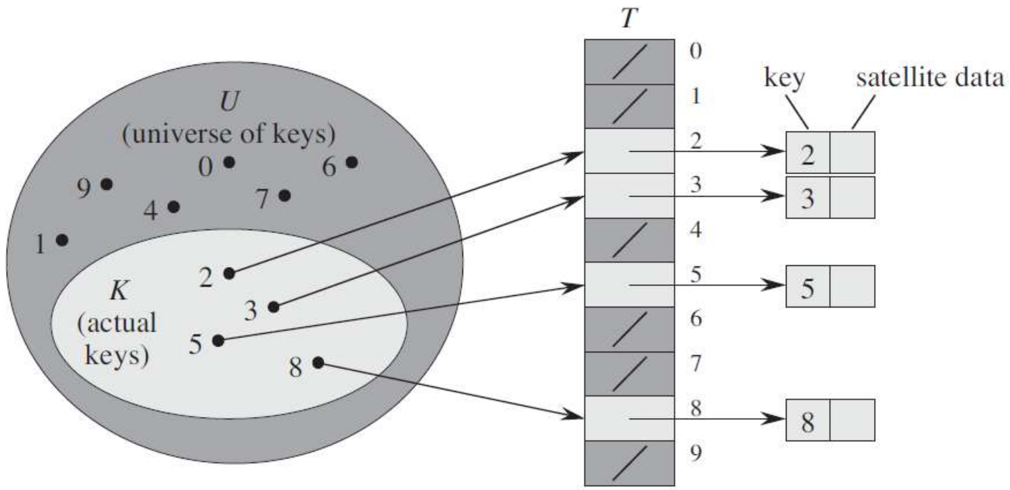


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

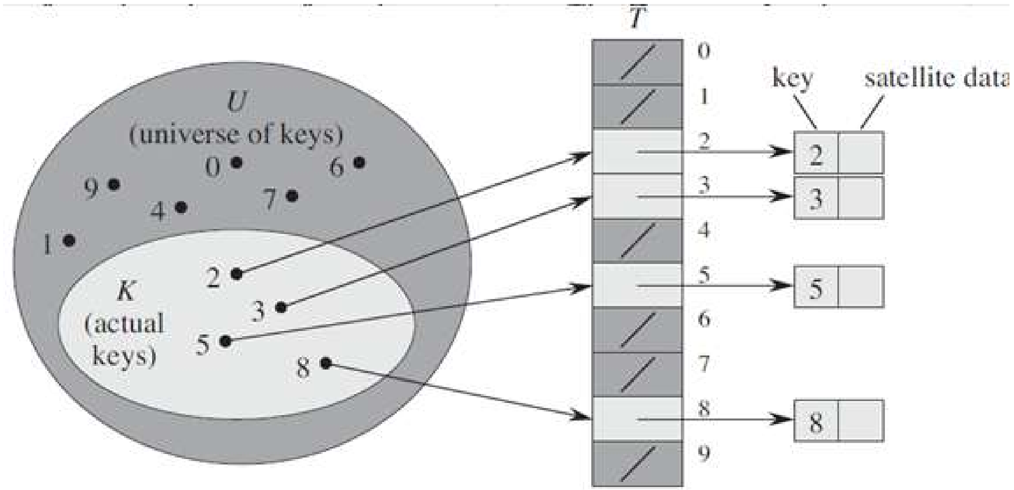
1 $T[x.key] =$ NIL

Each of these operations takes only $O(1)$ time.

Dinamic Set with Direct Address Table

- Suppose that a dynamic set S is represented by a direct-address table T of length m.
- Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure

We start with the bottom of the table (the largest element) and scan the table backwards until we find a slot that contains an element. The worst case performance is $\Theta(m)$



- The downside of direct addressing is obvious: if the universe U is large, storing a table T of size |U| may be impractical, or even impossible, given the memory available on a typical computer.
- Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

Hashing

- With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot h(k); that is, we use a ***hash function*** h to compute the slot from the key k.
- The hash function reduces the range of array indices and hence the size of the array.
- Instead of a size of |U| , the array can have size m.

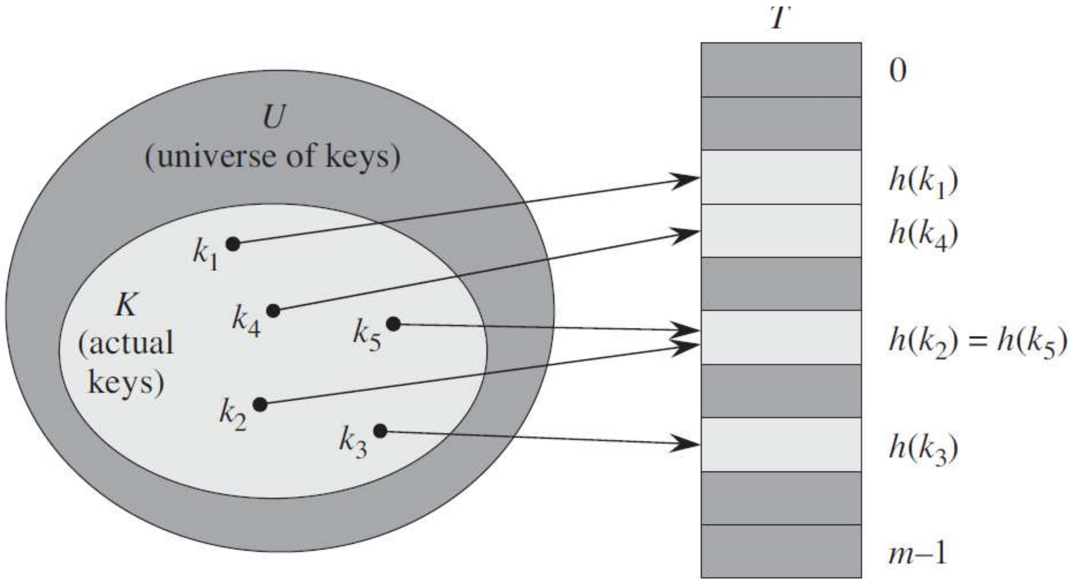


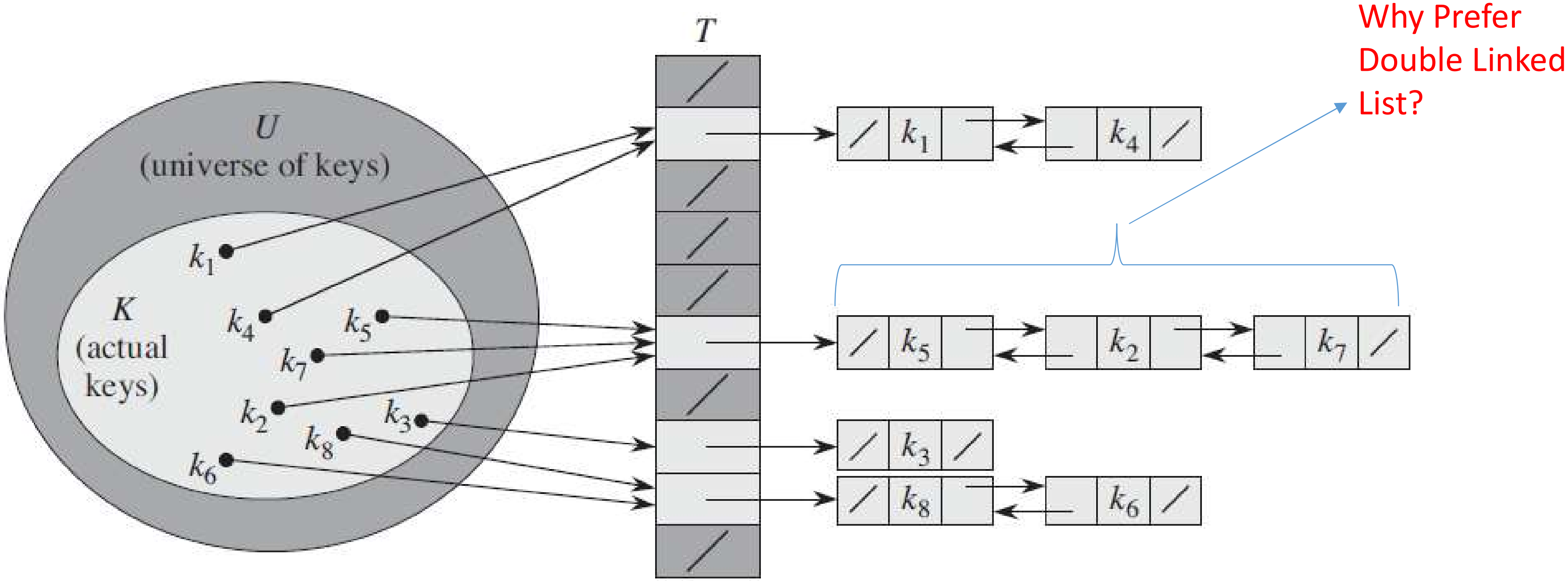
Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

Collision

- There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, we have effective techniques for resolving the conflict created by collisions.
- Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h.
- One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number.
- While a well designed, "random"-looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

Collision resolution by chaining

- In ***chaining***, we place all the elements that hash to the same slot into the same linked list.



- Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

The worst-case running time for insertion is O(1)

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

The worstcase running time is proportional to the length of the list O(n)

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

The worst-case running time for insertion is O(1)

What if we use single linked list?

Analysis of hashing with chaining

- Lets see some examples