Data Structures & Algorithms for Problem Solving (CS1.304)

Lecture # 03-04

Avinash Sharma

Center for Visual Information Technology (CVIT),

IIIT Hyderabad

Organization (today's lecture)

1. LINKED LIST

UNDERSTAND BASICS

2. LINKED LIST APPLICATIONS

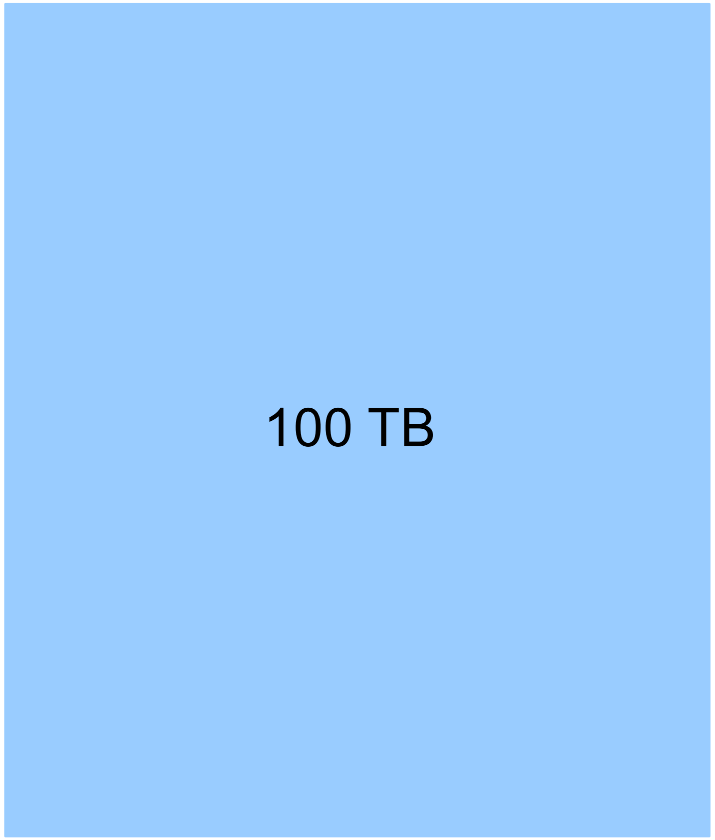
HOW TO FORMULATE ?

3. ISSUES

UNDERSTAND BASICS

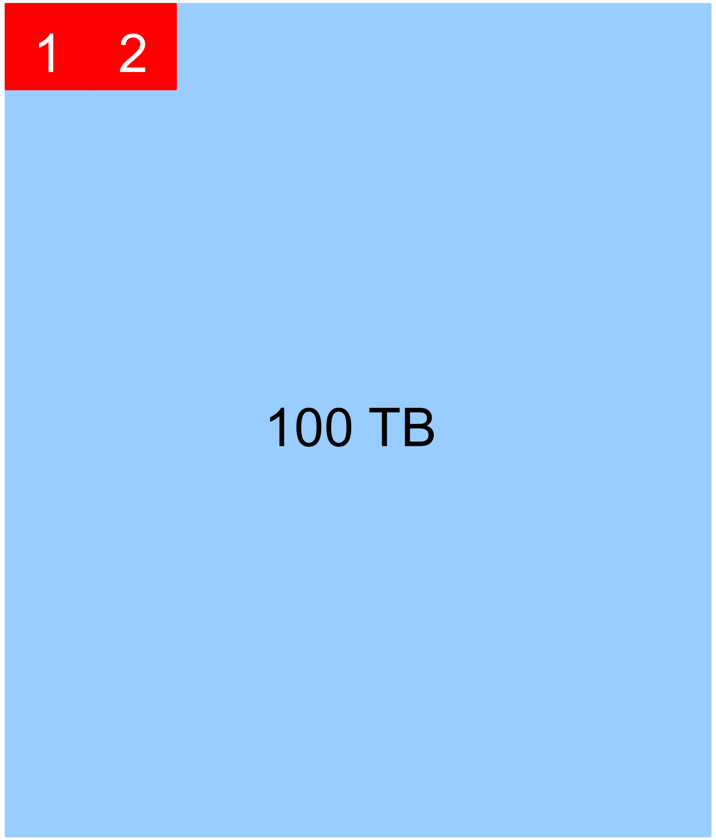
Motivation

- Change to disk storage solutions!!!
- Say we own a storage disk.
  - We have a huge amount of storage
  - Allot space to files.
- How should we arrange our storage?
- How should we allot space to files?



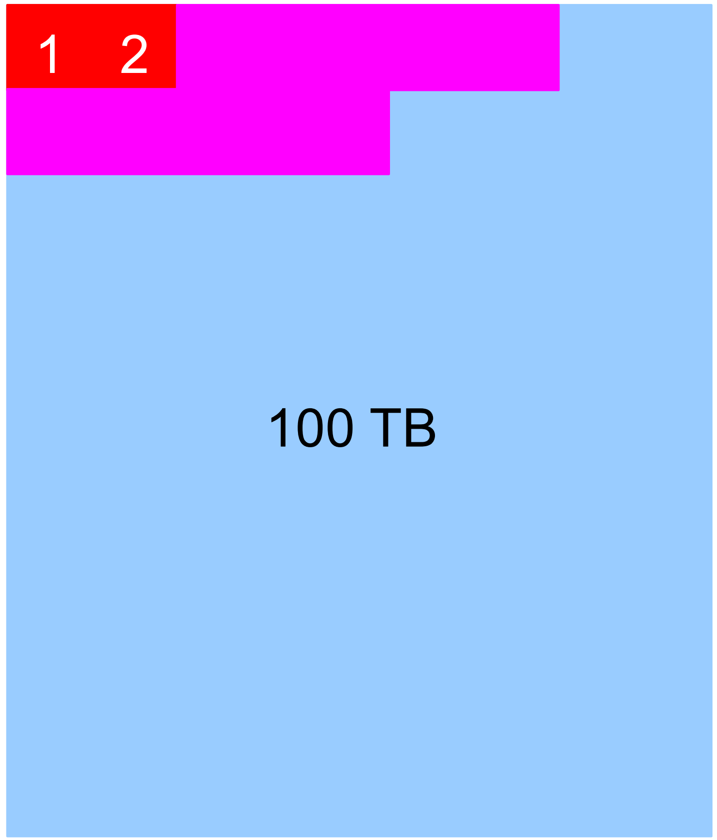
Motivation

- Suppose that from an initially empty state, File 1 asks for 100 MB of storage.
- It is given some space in the first row.
- Now, a second file needs some 100 MB.
- It is given the next available space.



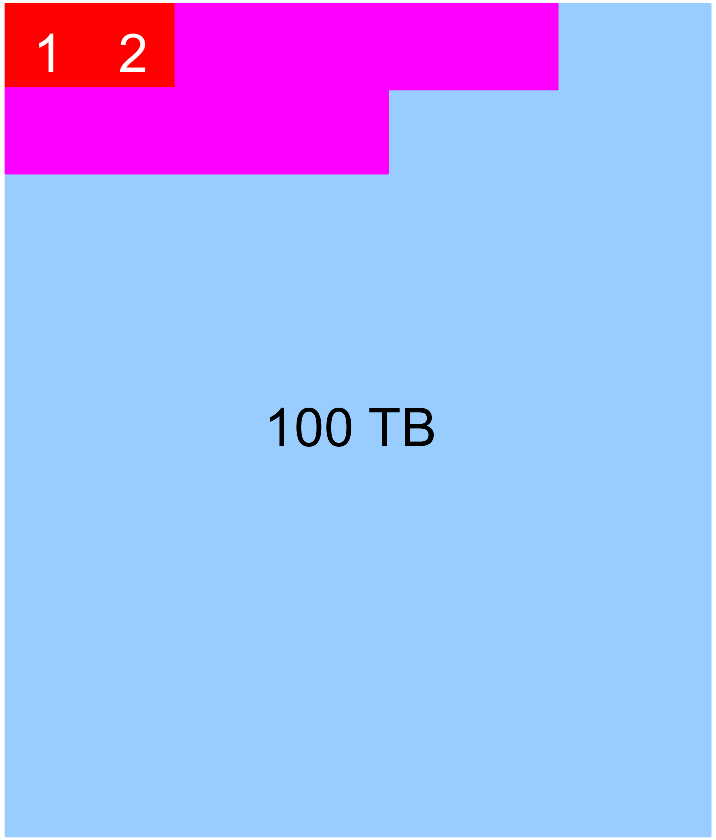
Motivation

- Similarly, some blocks are filled by clients.
- We also need a way to remember the area where we allot to each file.
  - But we will not worry too much about that.
- Now, the first file wants more space.
- Where should we allot more space?



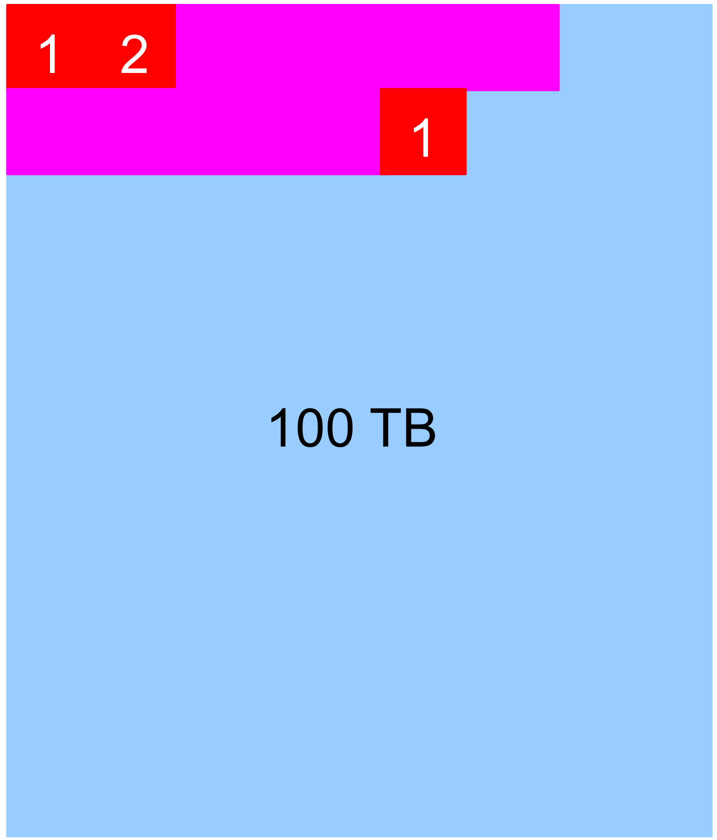
Motivation

- Option 1 : Contiguous allocation
  - Contiguous space for every user.
  - But, may have to move all other alloted users.
  - Very costly to do..
  - Think of further requests from this user for more space.



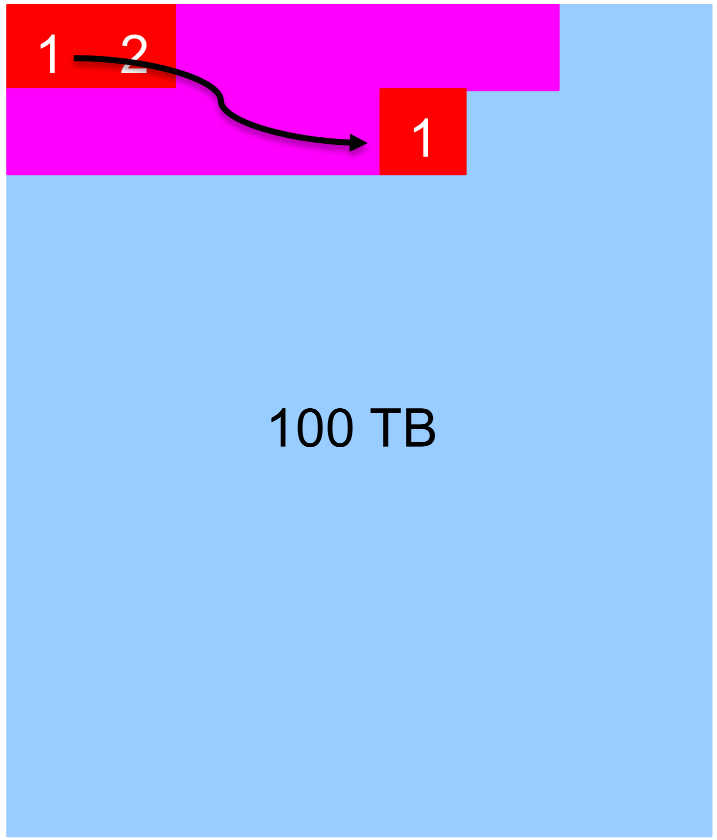
Motivation

- Ideal solution properties
  - Little update
  - No restriction on future requests of the same user or a different user.
- Can we allot pieces at different places?
- Problems
  - how to know what all pieces belong to a given user?



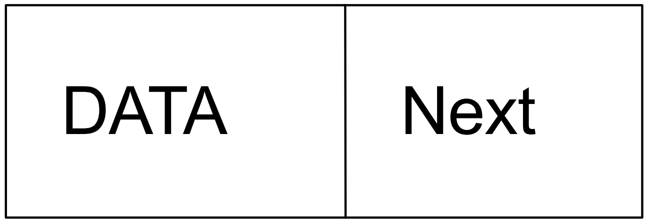
A Novel Solution

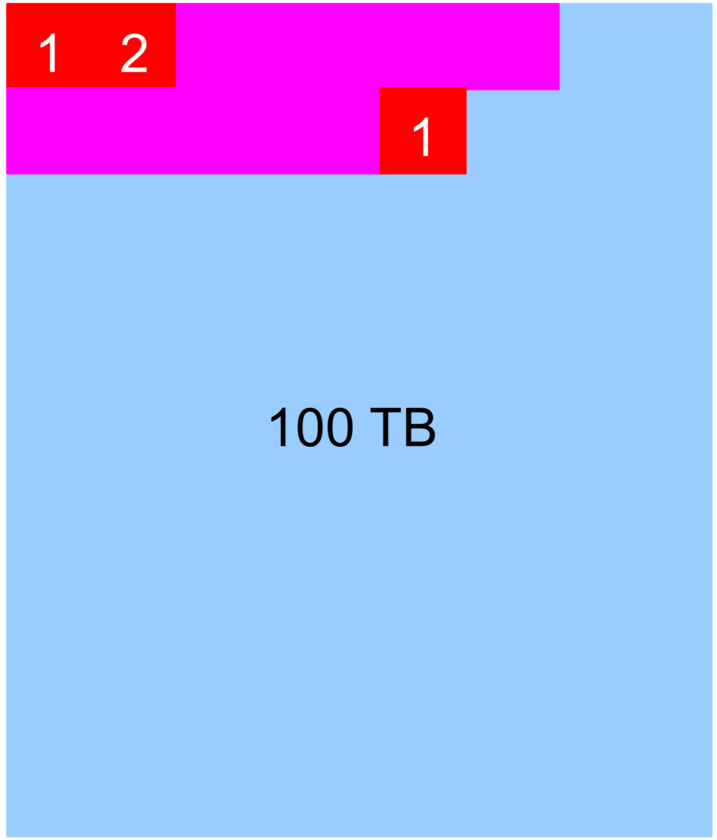
- At the end of every allocation, we leave some space to note down the details of the next allocation.
  - Empty for the last allocation
- Now, a user can know all the pieces of a file he owns by simply
  - starting from the first allocated piece
  - Find out if he has more pieces
  - Stop at the last piece



A Novel Solution

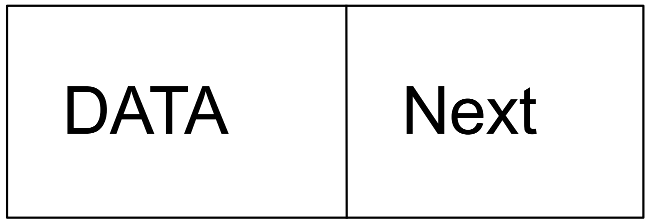
- The solution we saw just now is not new to Computer Science.
- The organization is called as a linked list.
  - Forms a part of data structures called pointer based data structures.



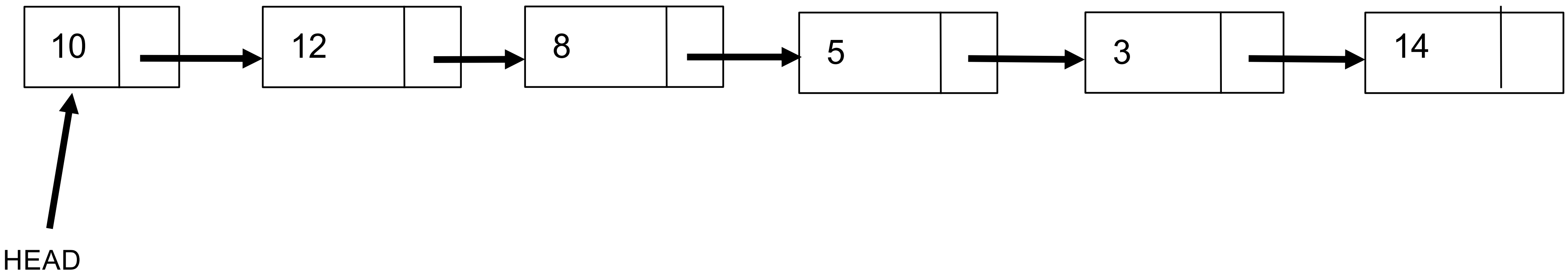


The Linked List

- The linked list is a pointer based data structure.
- Each node in the list has some data and then also indicates via a pointer the location of the next node.
  - Some languages call the pointer also as a reference.



The Linked List



- How to access a linked list?
  - Via a pointer to the first node, normally called the head.
- The figure above shows an example of representing a linked list.

Basic Operations

- Think of the array. We need to be able to:
  - Add a new element
  - Remove an element
  - Print the contents
  - Find an element
- Similarly, these are the basic operations on a linked list too.

Basic Operations

- To show the implementation, we assume:
  - the language supports pointers.
  - A C-like syntax.
  - A structure shown below.
  - Assume that for now, data is integers.

```
struct node
{
int data;
struct node *next;
}
```

Basic Operations

- Find Operation

```
Algorithm Find(x)
begin
temphead = head;
while (temphead != NULL) do
        if temphead ->data == x then
                return temphead;
        temphead = temphead ->next;
end-while
return NULL;
end
```

Basic Operations

- Print Operation

```
Algorithm Print()
begin
temphead = head;
while (temphead != NULL)
	Print(temphead ->data);
	tempead = temphead ->next;
end-while
end
```

Basic Operations

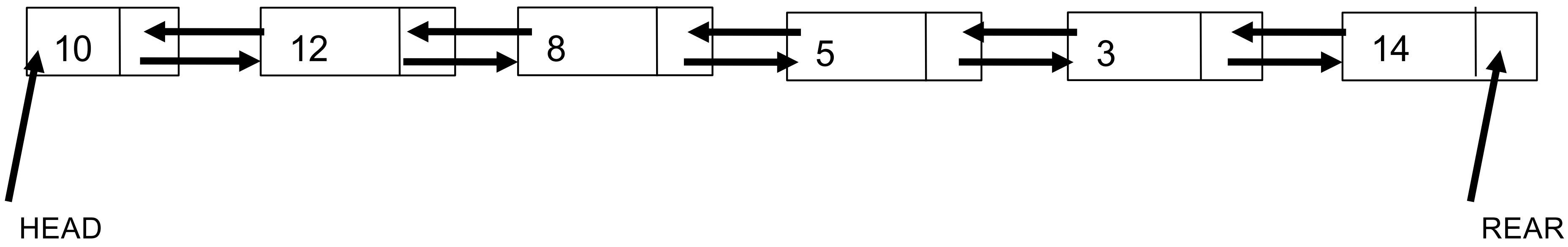
- To insert, where do we insert?
- Several options possible
  - insert at the beginning of the list
  - insert at the end
  - insert before/after a given element.
- Each applicable in some setting(s).
- We'll show insert at the front.
- Need to adjust the head pointer.

```
Algorithm Insert(item)
begin
  temphead = head;
  newnode = new node;
  newnode->data = item;
  newnode->next = temphead;
  head = newnode;
end
```

Basic Operations

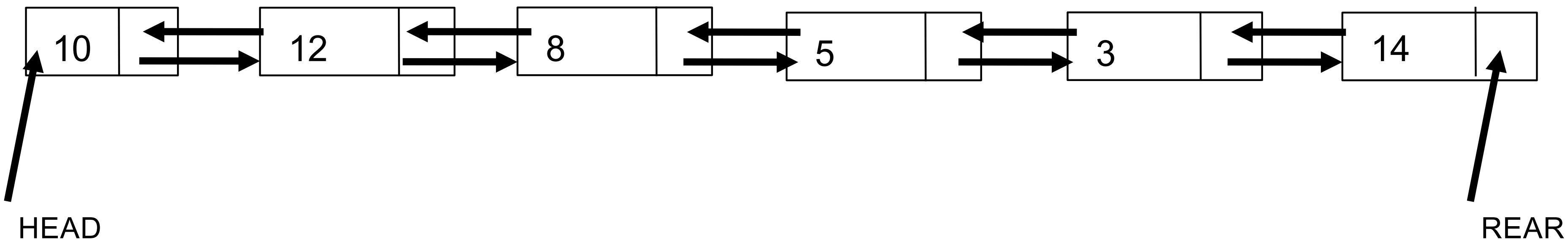
- Remove also has different possibilities.
  - Remove from the front
  - Remove before/after a given element.
  - Remove an existing element.
- Turns out each has application in some setting.
  - We'll see a few applications

Variation to a Linked List



- There are several variations to the (singly) linked list.
- Sometimes a doubly linked list is used.
  - Each node points to the predecessor as well as its successor.
  - Has two special pointers – head and rear

Advantages of Doubly Linked List

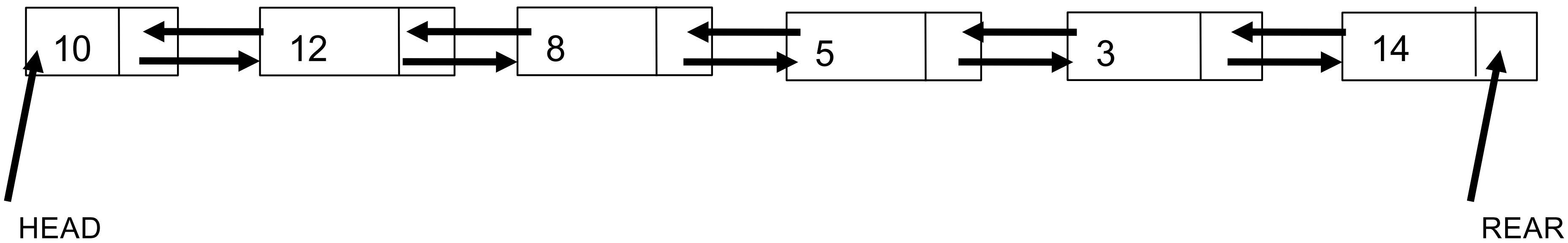


- **1)** It can be traversed in both forward and backward direction.

  2) The delete operation is more efficient if pointer to the node to be deleted is given.

  3) We can quickly insert a new node before a given node.

Doubly Linked List



- Write routines to insert and delete from a doubly linked list.
- We want to insert after a given element from the head, and delete a given item.
- Todo in class.

Application I – A Stack using Linked List

- One of the limitations of array based stack implementation is that we have to fix the maximum size of the stack.
  - The source of this limitation is that we had to specify the size of the array up front.
  - Using a dynamic data structure, we can remove this limitation.

Application I – A Stack using a Linked List

- A stack is a last-in-first-out based data structure.
- When using a linked list to implement a stack, we should
  - know how to translate push() and pop() of stack to linked list operations.
- We now would be seeing an implementation of an ADT using another ADT.

Application I – A Stack using a Linked List

- The push() operation can simply be translated to an insert at the beginning of the list.
- This suggests that pop would simply be translated to a remove operation at the front of the list.
- Does this keep the LIFO order?
  - Check it.

Application II – A Queue using a Linked List

- Another popular data structure is the *queue*.
- It maintains a first-in-first-out order.
- An array based implementation has a few drawbacks.
- We will use a linked list to implement queue operations.

Application II – A Queue using a Linked List

- Which kind of linked list to use?
- A doubly linked list may help.
  - It has a head and a rear identical to the front and rear of a queue.
- Can then translate queue operations Insert and Delete into insert and remove operations on a doubly linked list.

Application III – Representing Polynomials

- Another application of linked lists is to represent polynomials.
- A polynomial is a sum of terms.
- Each term consists of a coefficient and a (common) variable raised to an exponent.
- We consider only integer exponents, for now.
- Example: $4x^3 + 5x - 10$.

Application III – Representing Polynomials

- How to represent a polynomial?
- Issues in representation
  - should not waste space
  - should be easy to use it for operating on polynomials.

Application III – Representing Polynomials

- Any case, we need to store the coefficient and the exponent.
- Option 1 – Use an array.
  - Index k stores the coefficient of the term with exponent k.
- Advantages and disadvantages
  - Exponent stored implicitly (+)
  - May waste lot of space. When several coefficients are zero (– –)
  - Exponents appear in sorted order (+)

Application III – Representing Polynomials

- Further points
  - Even if the input polynomials are not sparse, the result of applying an operation to two polynomials could be a sparse polynomial. (--)
- How can we use a linked list ?

```
struct node
{
float coefficient;
int exponent;
struct node *next;
}
```

Application III – Representing Polynomials

- Each node of the linked list stores the coefficient and the exponent.
- Should also store in the sorted order of exponents.
- How can a linked list help?
  - Can only store terms with non-zero coefficients.
  - Does not waste space.
  - Need not know the terms in a result polynomial apriori. Can build as we go.

```
struct node
{
float coefficient;
int exponent;
struct node *next;
}
```

Application III – Operations on Polynomials

- Let us now see how two polynomials can be added.
- Let P1 and P2 be two polynomials.
  - stored as linked lists
  - in sorted (decreasing) order of exponents
- The addition operation is defined as follows
  - Add terms of like-exponents.

Application III – Operations on Polynomials

- We have P1 and P2 arranged in a linked list in decreasing order of exponents.
- We can scan these and add like terms.
  - Need to store the resulting term only if it has non-zero coefficient.
- The number of terms in the result polynomial P1+P2 need not be known in advance.
- We'll use as much space as there are terms in P1+P2.

Application III – Operations on Polynomials

- Let us consider multiplication
- Can be done as repeated addition.
- So, multiply P1 with each term of P2.
- Add the resulting polynomials.

- Develop the pseudocode in class...

Application IV – Matrix Multiplication

- Consider another problem described as follows.
- The multiplication of two matrices A and B is understood as follows.
- For each i and j, C[i,j] = Σ_k A[i,k].B[k,j].

$$\underset{A}{\begin{bmatrix} 3 & 2 \\ 1 & 1 \\ 5 & 2 \end{bmatrix}} \times \underset{B}{\begin{bmatrix} 2 & 1 & 3 \\ 3 & 3 & 1 \end{bmatrix}} = \underset{C}{\begin{bmatrix} 12 & 9 & 11 \\ 5 & 4 & 4 \\ 16 & 11 & 17 \end{bmatrix}}$$

Application IV – Matrix Multiplication

- If A and B are sparse, there are several issues in matrix multiplication if A, B, and C are stored as arrays.
  - Storage /Retrieval, Compatibility of indices
- Alternate storage models for sparse matrices exist.

Row	Col	Val
1	2	10
1	3	12
2	1	1
2	3	2

Row	Col	Val
1	1	2
1	2	5
2	2	1
3	1	8

A

0	10	12
1	0	2
0	0	0

B

2	5	0
0	1	0
8	0	0

Application IV – Matrix Multiplication

$$\begin{bmatrix} 0 & 10 & 12 \\ 1 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 5 & 0 \\ 0 & 1 & 0 \\ 8 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 96 & 10 & 0 \\ 18 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Application IV – Matrix Multiplication

- To multiply A and B, get each row of A and each column of B multiply element-wise and sum to get one element of C.
- Not easy if sparse matrix are stored as sorted list. Can we do it efficiently ?

Row	Col	Val
1	2	10
1	3	12
2	1	1
2	3	2

Row	Col	Val
1	1	2
1	3	8
2	1	5
2	2	1

0	10	12
1	0	2
0	0	0

A

2	0	8
5	1	0
0	0	0

B^T

Matrix Multiplication – Column-Row Formulation

$$\overset{A}{\begin{bmatrix} 3 & 2 \\ 1 & 1 \\ 5 & 2 \end{bmatrix}} \times \overset{B}{\begin{bmatrix} 2 & 1 & 3 \\ 3 & 3 & 1 \end{bmatrix}} = \overset{C}{\begin{bmatrix} 12 & 9 & 11 \\ 5 & 4 & 4 \\ 16 & 11 & 17 \end{bmatrix}}$$

$$\begin{bmatrix} 3 \\ 1 \\ 5 \end{bmatrix} * \begin{bmatrix} 2 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 9 \\ 2 & 1 & 3 \\ 10 & 5 & 15 \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} * \begin{bmatrix} 3 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 6 & 2 \\ 3 & 3 & 1 \\ 6 & 6 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 12 & 9 & 11 \\ 5 & 4 & 4 \\ 16 & 11 & 17 \end{bmatrix}$$

Matrix Multiplication – Row-Row Formulation

$$\underset{A}{\begin{bmatrix} 3 & 2 & 0 \\ 1 & 1 & 2 \end{bmatrix}} \times \underset{B}{\begin{bmatrix} 2 & 1 & 3 \\ 3 & 3 & 1 \\ 2 & 2 & 1 \end{bmatrix}} = \underset{C}{\begin{bmatrix} 12 & 9 & 11 \\ 9 & 8 & 6 \end{bmatrix}}$$

$$3 \times \begin{bmatrix} 2 & 1 & 3 \end{bmatrix} + 2 \times \begin{bmatrix} 3 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 12 & 9 & 11 \end{bmatrix}$$

$$1 \times \begin{bmatrix} 2 & 1 & 3 \end{bmatrix} + 1 \times \begin{bmatrix} 3 & 3 & 1 \end{bmatrix} + 2 \times \begin{bmatrix} 2 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 9 & 8 & 6 \end{bmatrix}$$

Application IV – Matrix Multiplication

- Now develop psuedocode for multiplying matrices using the Row-Row formulation OR sparse list transpose based row-row formulation

Application IV – Matrix Multiplication

- There are several other applications for linked lists.
- Mostly in places where one needs a dynamic ability to grow/shrink.
- However, one has to keep the following facts in mind.
- How are they managed on most present systems?

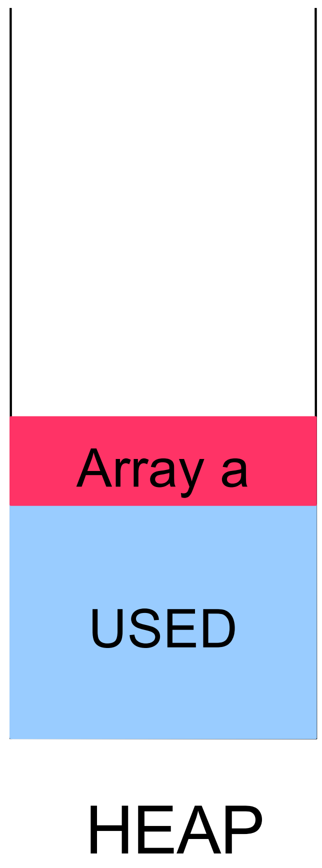
Linked List

- How are they managed on most present systems?
- To understand, consider where arrays are stored?
  - At least in C and UNIX, depends on the type of the declaration.
  - A static array is stored on the program stack.
  - Example: int a[10];
- There is a memory called the heap.
  - Dynamically specified arrays are stored on the heap.
  - Example follows.

Heap Allocation

```
int *a;
.
.
a = (int *) malloc(100);
.
```

- ‘Array a’ alloted on the heap.
- But given contiguous space.
- Hence, a+20 can be used to access a[5] also.

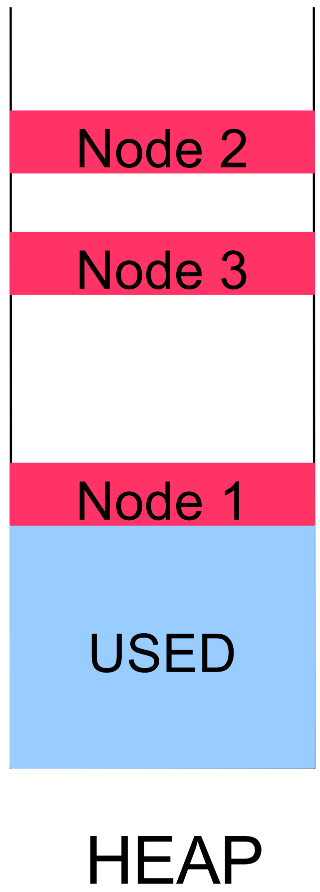


Heap Allocation

- Such a contiguous allocation:
  - benefits cache behavior (++)
  - cannot alter the size of the array later (--)
  - easy addressing (+)
- Modern compilers and hardware actually use techniques such as pre-fetching so that the program can experience more cache hits.
- This is important as memory access times are constantly increasing relative to processor speed.

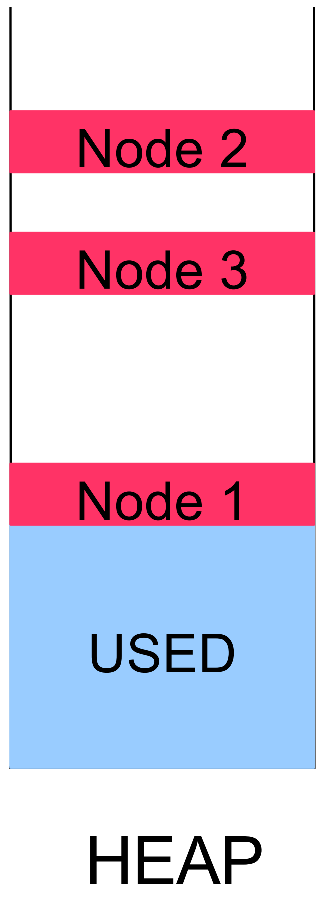
Heap Allocation

- Nodes added to the linked list are always alloted on the heap.
  - There is always a malloc call before adding a node.
  - Example below.



Linked List

- What does the next really store?
- The address of the next node in the list.
- This could be anywhere in the heap.



Implications of Linked List

- Cache not very helpful.
  - Cannot know where the next node is.
- No easy pre-fetching.
- When programming for performance, this can be a big penalty.
  - Especially critical software such as embedded systems software.

Thank You