Further Data Structures

- The story so far
  - We understand the notion of an abstract data type.
  - Saw some fundamental operations as well as advanced operations on arrays.
  - Saw how restricted/modified access patterns on even arrays have several applications.
- This week we will
  - study a data structure that can grow dynamically
  - its applications

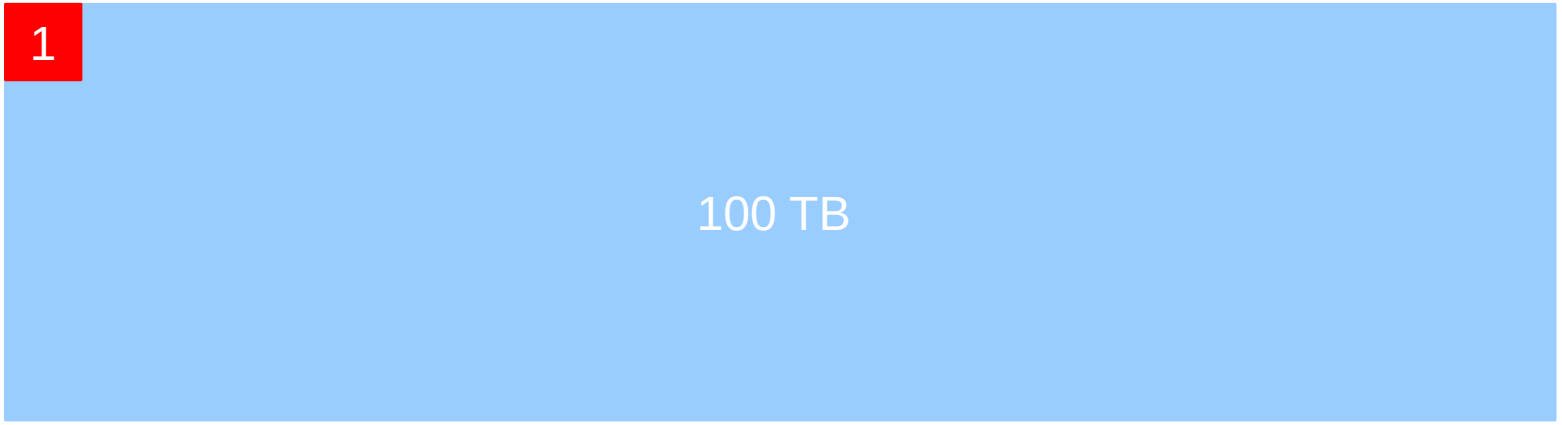
Motivation

- How files are stored on a disk.
  - We have a huge amount of storage space.
  - Assign it to various files.
- How should we arrange our storage?
- How should we allot space to files?

Motivation

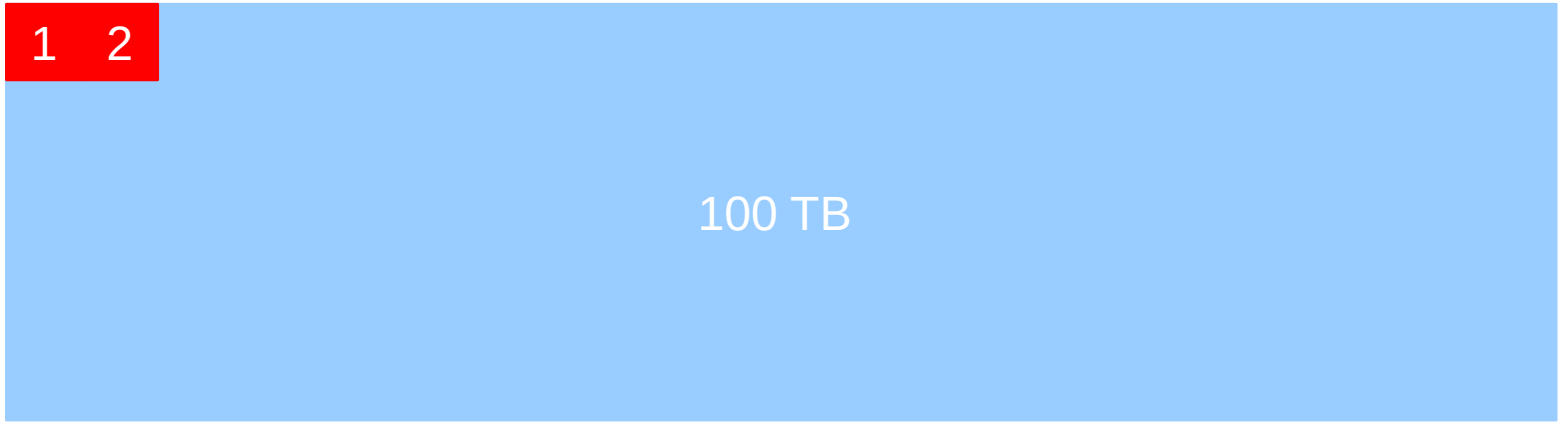
- Our comments apply to most storage arrangement patterns.
- So we'll focus only on how to allot storage.

Motivation



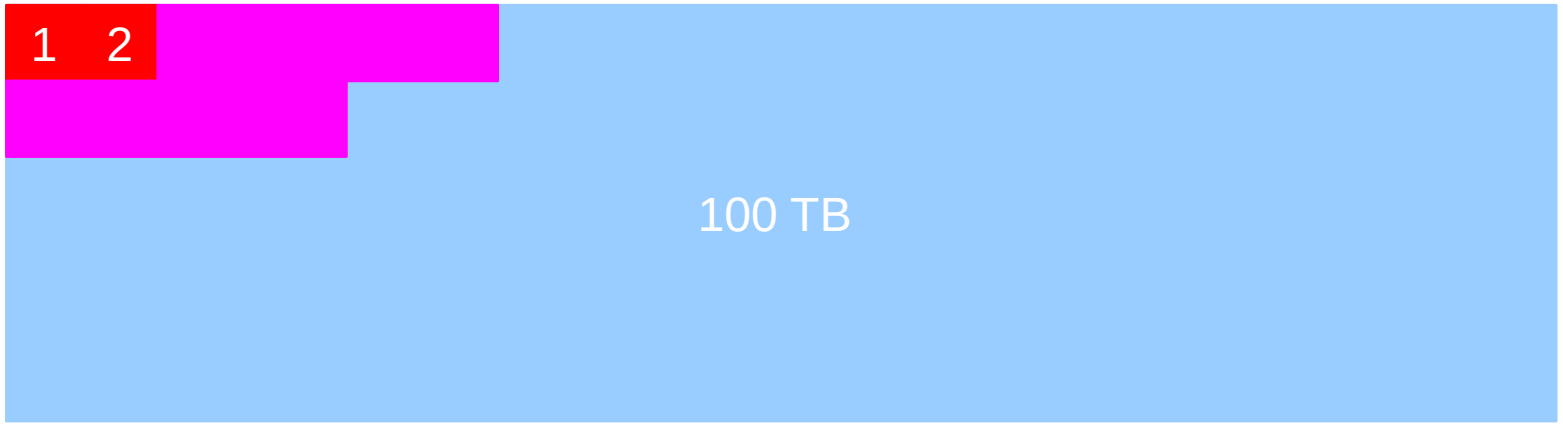
- Suppose that from an initially empty state, file 1 asks for 100 MB of storage.
- File1 given some space in the first row.

Motivation



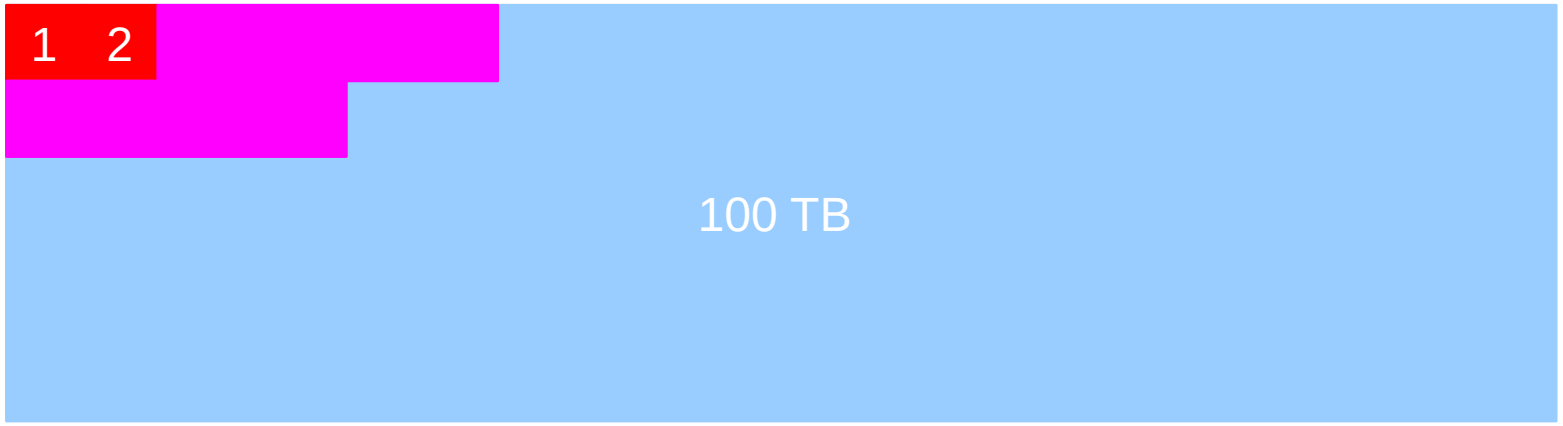
- Now, a second file needs some 100 MB.
- This file is given the next available space.

Motivation



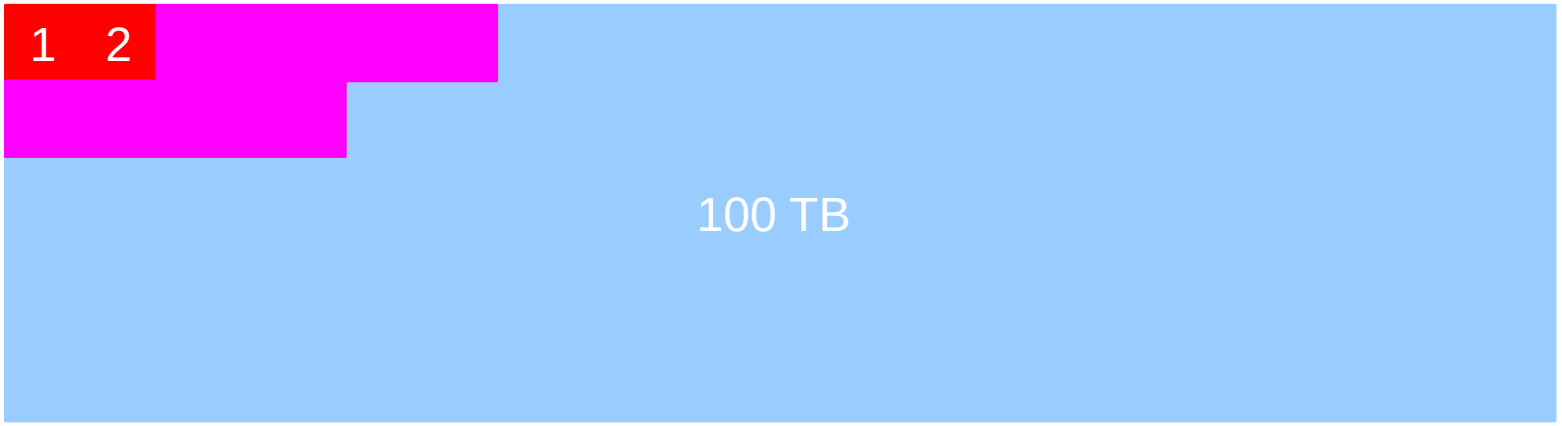
- Similarly, some space is filled by files.
- We also need a way to remember the area where we allot to each file.
  - But we will not worry too much about that.

Motivation



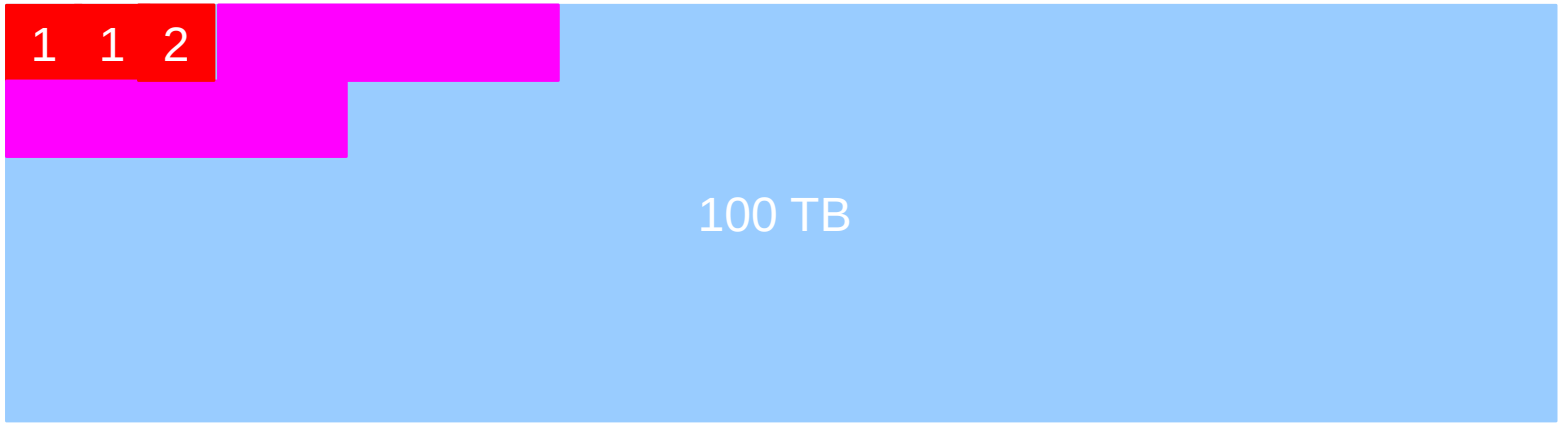
- Now, the first file wants more space.
- Where should we allot more space?

Motivation



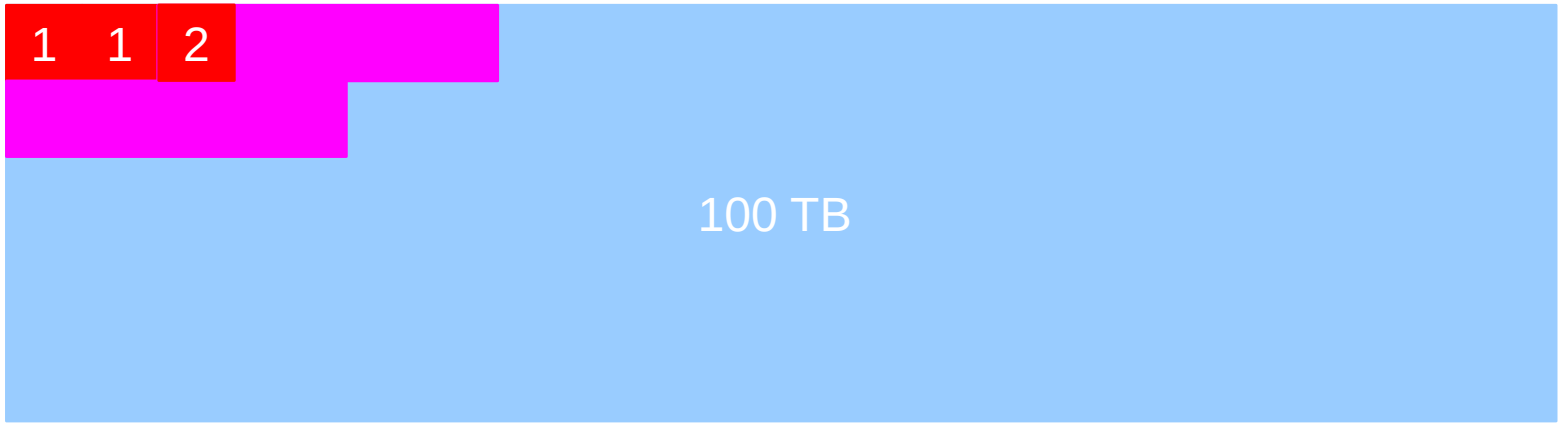
- Option 1 : Contiguous allocation
  - Contiguous space for every file.
  - But, may have to move all other alloted files.
  - Very costly to do..
  - Think of further requests from this user for more space.

Motivation



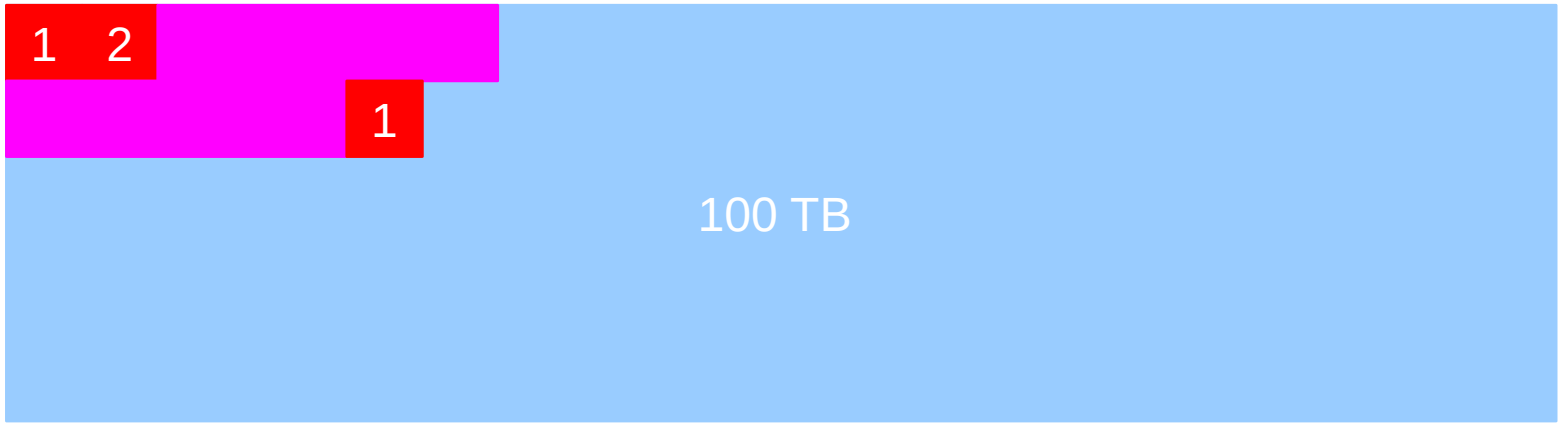
- Option 1 : Contiguous allocation
  - Contiguous space for every file.
  - But, may have to move all other alloted files.
  - Very costly to do..
  - Think of further requests from this user for more space.

Motivation



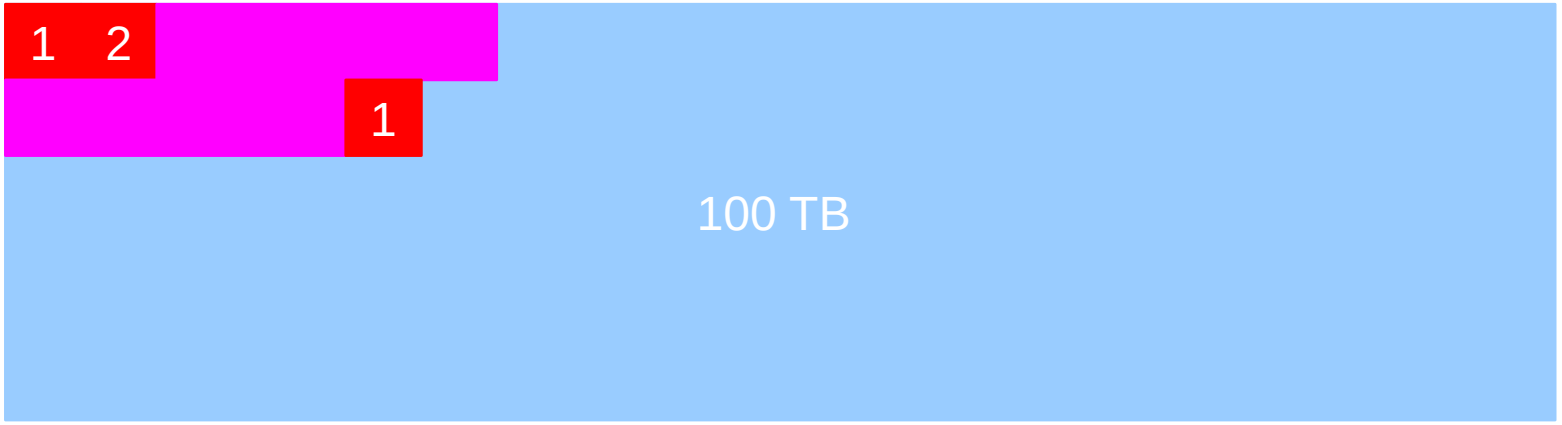
- Ideal solution properties
  - Little update
  - No restriction on future requests of the same file or a different file.

Motivation



- Can we allot pieces at different places?
- Problems
  - how to know what all pieces belong to a given user?

A Novel Solution



- Imagine that at the end of every allocation, we leave some space to note down the details of the next allocation.
  - For the last allocation, this space could be empty.

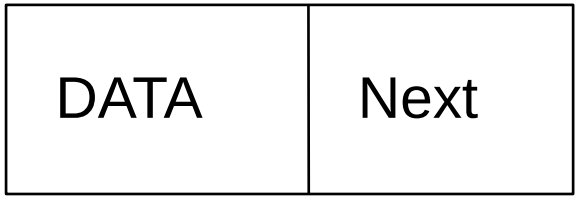
A Novel Solution

- Now, a file has multiple pieces of storage. Can know all the pieces of the file by simply
  - starting from the first alloted piece
  - Find out if there are more pieces
  - Stop at the last piece

A Novel Solution

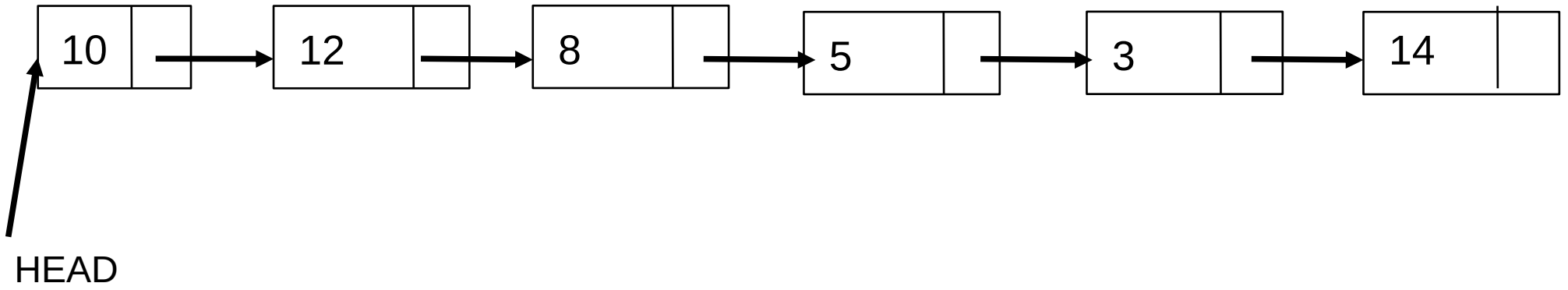
- The solution we saw just now is not new to Computer Science.
- The organization is called as a linked list.
  - Forms a part of data structures called pointer based data structures.

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 node structure is as shown in the figure.

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.

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

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

Basic Operations

```
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
```

- We'll start with operation Find.

Basic Operations

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

- Algorithm to print the contents shown above.

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).

Basic Operations

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

- We'll show insert at the front.
- Need to adjust the head pointer.

Basic Operations

- Practice Problems:
  - Write pseudocode for inserting an element at the end of the list.
  - Write pseudocode for inserting an element after a given element. As an example, if the list is 12->38->11->42, on calling Insert(15, 38), the new list should be 12->38->15 ->11->42.

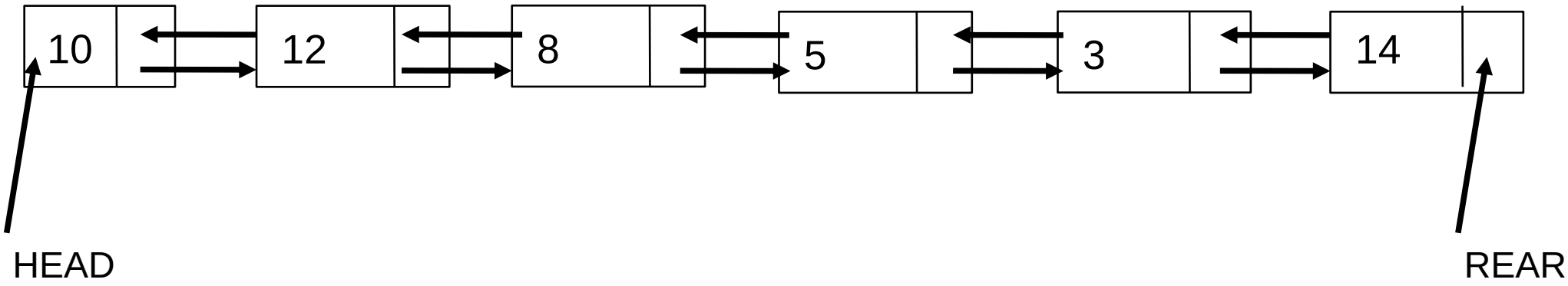
Basic Operations

- 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

Practice Problems

- Write pseudocode for the three different remove operations.

Variations to a Linked List



- There are several variations to the linked list.
- The one seen so far is called as the singly linked list.
- Sometimes people use a doubly linked list.
  - Each node points to the predecessor as well as its successor.
  - Has two special pointers – head and rear

Application I – A Stack using a Linked List

- One of the limitations of our stack implementation earlier 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.
  - The details follow.

A Stack Using a Linked List

- Recall that a stack is a last-in-first-out based data structure.
- When using a linked list to support 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.

Push() and Pop() on Stack

- 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 data structure we saw earlier 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.

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 3 – Polynomials

- Another application of linked lists is to 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 3 – Polynomials

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

Application 3 – 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 implicity (+)
  - May waste lot of space. When several coefficients are zero (– –)
  - Exponents appear in sorted order (+)

Application 3 – 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. (--)

Application 3 – 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. (--)
  - Can you find such an example for addition?

Application 3 – 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. (--)
  - Can you find such an example for addition?
  - For multiplication?

Application 3 – Polynomials

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

- Can we use a linked list?
- Each node of the linked list stores the coefficient and the exponent.
- Should also store in the sorted order of exponents.
- The node structure is as follows:

Application 3 -- Polynomials

- 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.

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.

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.

Further Operations

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

Further Operations

- Write the pesudocode for this operation.
- You can imagine an AddPolynomial(P1, P2) routine that adds two polynomials and returns another polynomial P3, all linked lists.

Linked Lists

- There are several other applications for linked lists.
- Mostly in places where one needs a dynamic ability to grow/shrink.
- Applications include also representing sparse matrices.
- However, one has to keep the following facts in mind.

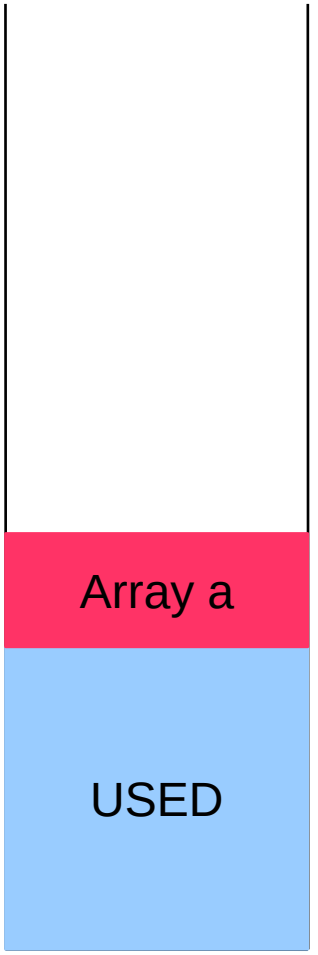
Linked Lists

- 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.

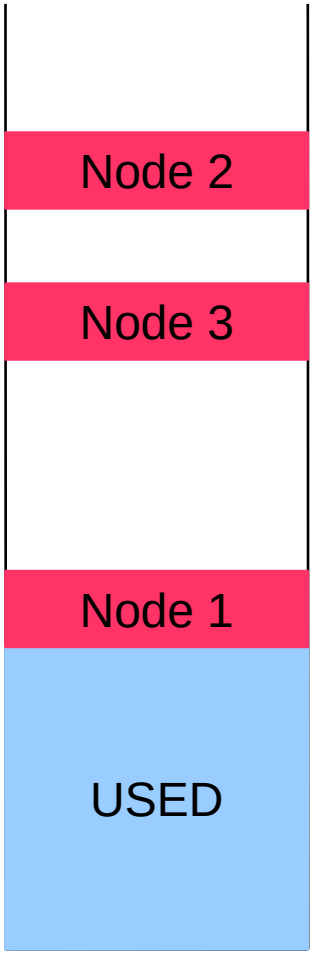


Heap Allocation

- 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.

How about a Linked List

- 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.
- See the earlier example.

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.