Cây tìm kiếm nhị phân
(Binary Search Tree)

Nội dung

I. Bài toán tìm kiếm

II. Khái niệm Cây nhị phân

III. Cấu trúc Cây TKNP

IV. Bài tập

I. Bài toán Tìm kiếm (Searching problem)

Bài toán

Input: Cho một tập ***S*** các phần tử, mỗi phần tử là một bộ gồm khóa-giá trị (key-value) và một khóa ***k*** bất kỳ.

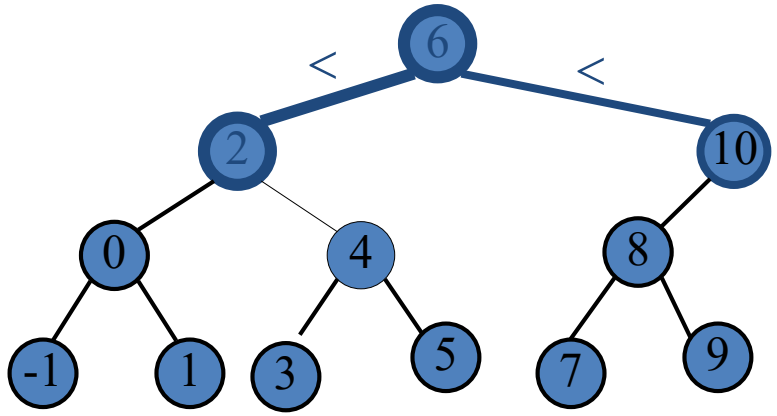
Output: Trong ***S*** có phần tử có khóa ***k*** hay không?

Khóa:

- Là thành phần (trường) dữ liệu để tìm kiếm
- Các khóa phải thỏa mãn một quan hệ thứ tự tổng thể.
- So sánh được xác định cho bất kỳ cặp khóa nào và quy tắc so sánh phải là phản xạ, phản đối xứng và bắc cầu.

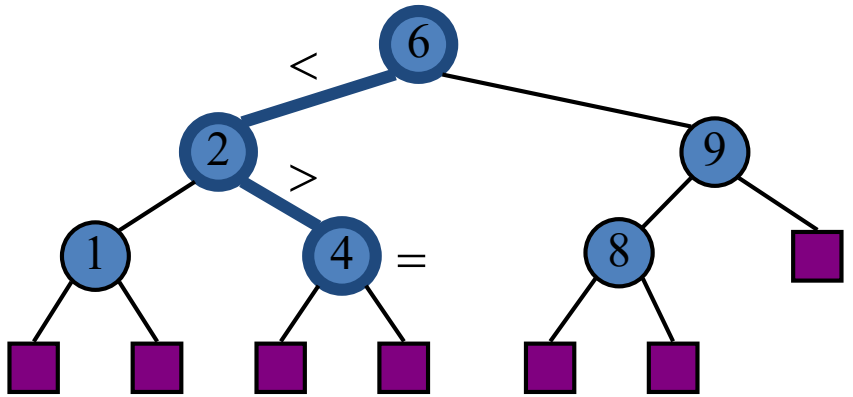
II. Khái niệm cây tìm kiếm nhị phân

- **Định nghĩa: cây tìm kiếm nhị phân là cây nhị phân thỏa mãn:**
  - Nút cha có khóa lớn hơn khóa của tất cả các nút của cây con bên trái và nhỏ hơn khóa của tất cả các nút của cây con bên phải.
  - Các nút con trái và phải cũng là cây tìm kiếm nhị phân

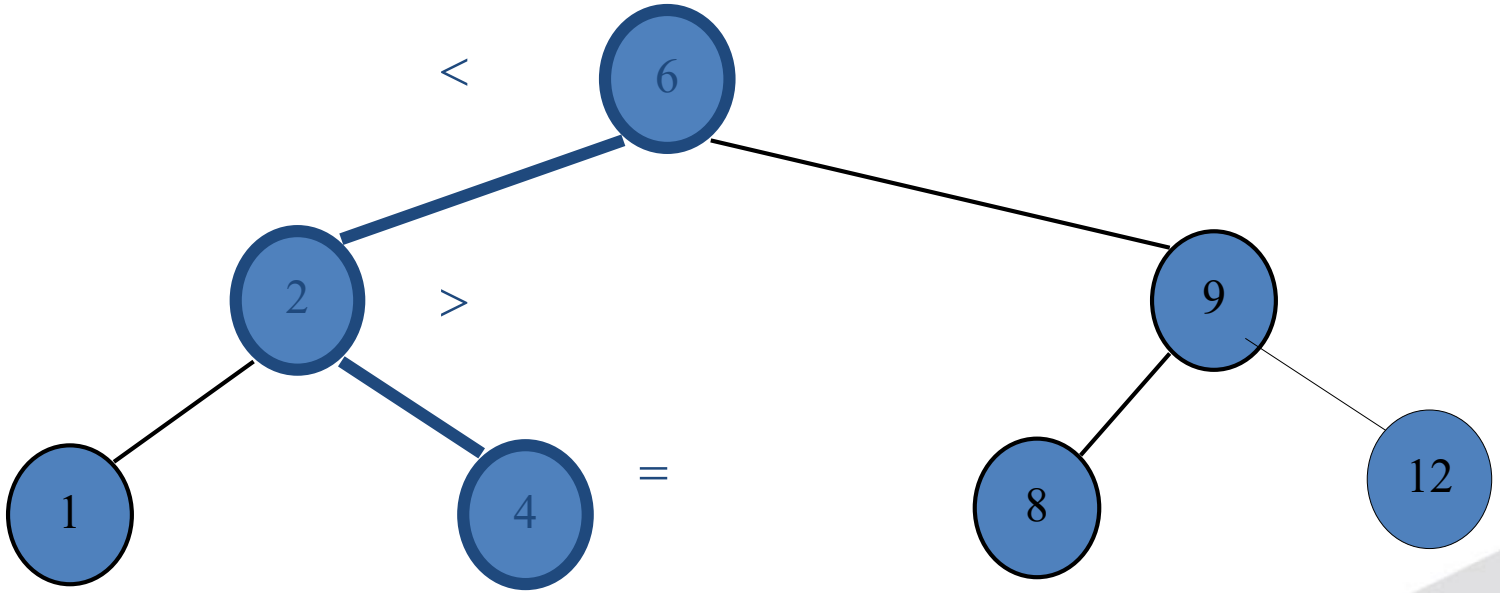


Ví dụ: Cây tìm kiếm nhị phân

Ví dụ Cây tìm kiếm nhị phân



find(4)



III. Cấu trúc Node biểu diễn cây nhị phân

– **Thuộc tính**
  - Keys *key*
  - T elem
  - Node *Parent
  - Node *Left
  - Node *Right

Chú ý: Keys là tập các giá trị được sắp có thứ tự

- Phương thức
  - Node *getParent()
  - Node *getLeft()
  - Node *getRight()
  - void setLeft(Node*)
  - void setRight(Node*)
  - void setParent(Node *)
  - int hasLeft()
  - int hasRight()
  - T getElem()
  - void setElem(T o)
  - void setKey(Keys k)
  - Keys getKey()

Cấu trúc của cây tìm kiếm nhị phân

- **Thuộc tính**
  - Node<Keys,T> * root
- **Các phương thức truy cập:**
  - Node<Keys,T> *root()
- **Phương thức**
  - int size()
  - int isEmpty()
  - int isInternal(Node<Keys,T>*)
  - int isExternal(Node<Keys,T>*)
  - int isRoot(Node<Keys,T>*)
  - void preOrder(Node<Keys,T>*)
  - void inOrder(Node<Keys,T>*)
  - void postOrder(Node<Keys,T>*)
  - Node<Keys,T>*search(Keys, Node<Keys,T>*)
  - Node<Keys,T>* insert(Keys, T)
  - void remove(Keys)

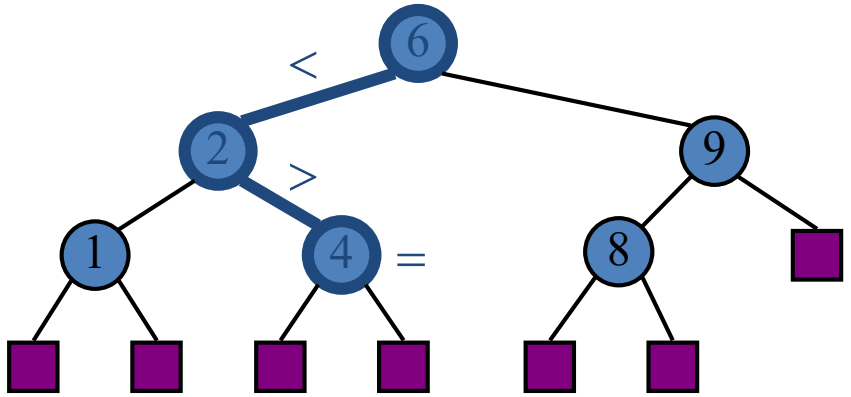
Thuật toán tìm kiếm

- **Tìm trong cây có nút có khóa bằng *k* không?**
- **Thuật toán**
  - Xuất phát từ nút gốc
  - So sánh giá trị khóa của nút gốc với k
    - Nếu giá trị của gốc =***k*** thì trả lại đ/c của nút đó và dừng lại
    - Nếu giá trị của nút gốc <***k*** thì tiếp tục tìm trên cây con phải
    - Nếu giá trị của nút gốc >***k*** thì tiếp tục tìm trên cây con trái
    - Quá trình tìm dừng lại khi tìm thấy hoặc phải tìm trên cây rỗng, trả lại địa chỉ của nút mà thuật toán dừng lại

Thuật toán giả mã

```
Node* search(Keys k, Node<T>* v)
   if(v==NULL)
            return NULL; //Ko tìm thấy
   else
          if (k < v->getKey())
             return search(k, v->getLeft());
          else if (k == v->getKey())
             return v;
          else // k > v->getKey()
             return search(k, v->getRight());
```

Ví dụ

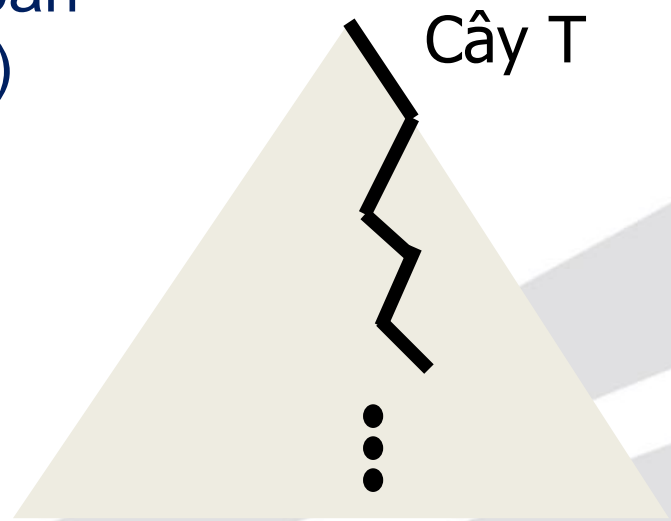


Tìm giá trị 4 trên cây:

- Gọi T.search(4, T.root())
- Gọi T.search(4, T.root()->getLeft()))
- Gọi T.search(4, T.root()->getLeft()->getRight())

Phân tích thời gian

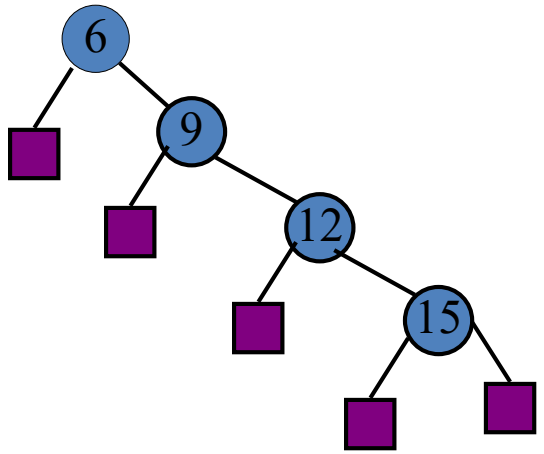
- **Thuật toán search**
  - Là thuật toán đệ qui,
  - Mỗi lần gọi đệ qui nó thực hiện một số phép toán cơ bàn không đổi, vậy một lần gọi đệ qui cần thời gian là O(1)
  - Thực hiện gọi đệ qui dọc theo các nút, bắt đầu từ nút gốc, mỗi lần gọi đệ qui nó đi xuống một mức.
  - Do đó số nút tối đa mà nó phải đi tới không vượt quá chiều cao *h* của cây.
  - Thời gian chạy là O(*h*)

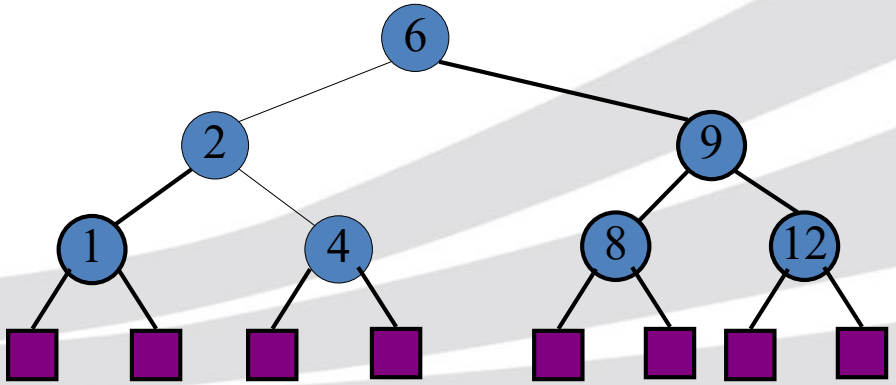


- Trong trường hợp xấu nhất cây có chiều cao bằng bao nhiêu?

Một số trường hợp

- **Cây chỉ có con trái hoặc con phải**
  - Chiều cao cây bằng số nút của cây
- **Cây hoàn chỉnh**
  - Chiều cao của cây là $\log_2 n$

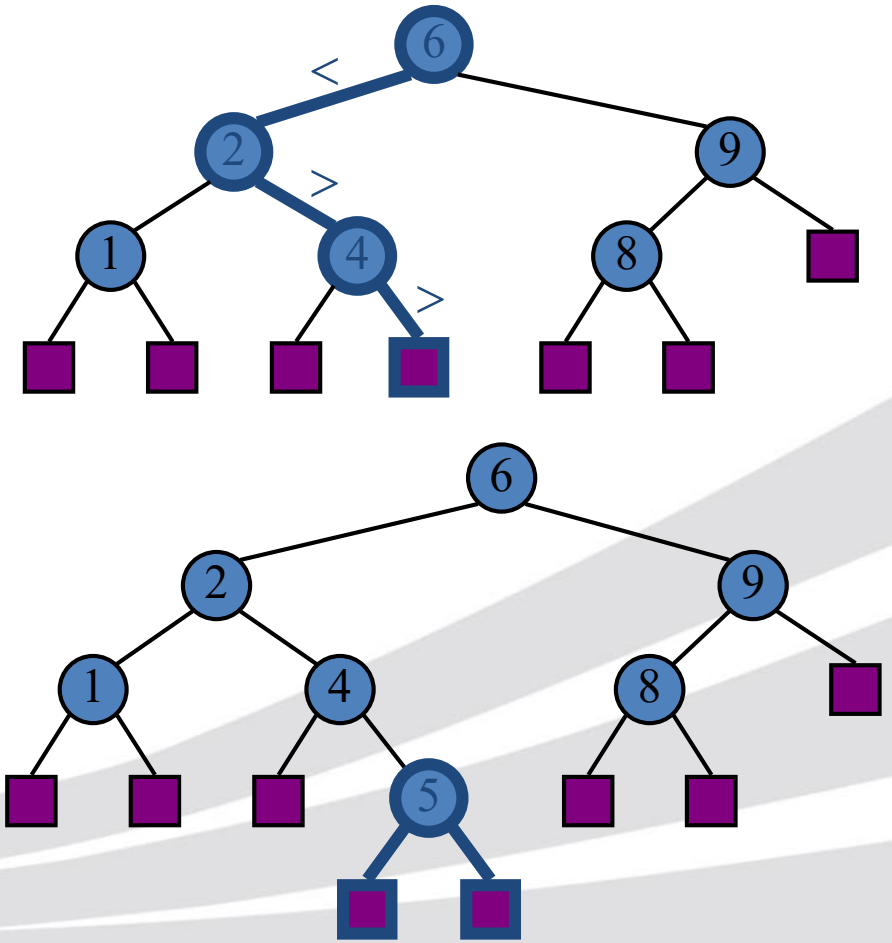




Bổ sung nút vào cây- Insertion

Sau khi bổ sung cây vẫn thỏa mãn tính chất cây TKNP

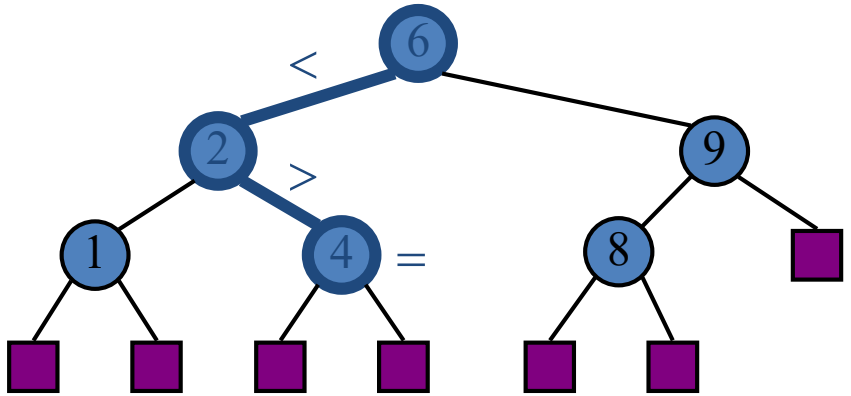
- **Bổ sung một nút với khóa *k* và value *x***
  - Thực hiện tìm kiếm k trên cây
- **Giả sử *k* không có trên cây**
  - Nếu cây rỗng thì gán nút cho gốc cây
  - Ngược lại, kiểm tra
    - Nếu khóa k< khóa nút gốc thì chén nút mới vào cây con bên trái
    - Nếu khóa k> khóa nút gốc thì chèn nút mới vào cây con bên phải



Ví dụ

- Xây dựng cây TKNP từ dãy số sau: 50, 30,80,32, 70,21,6,9

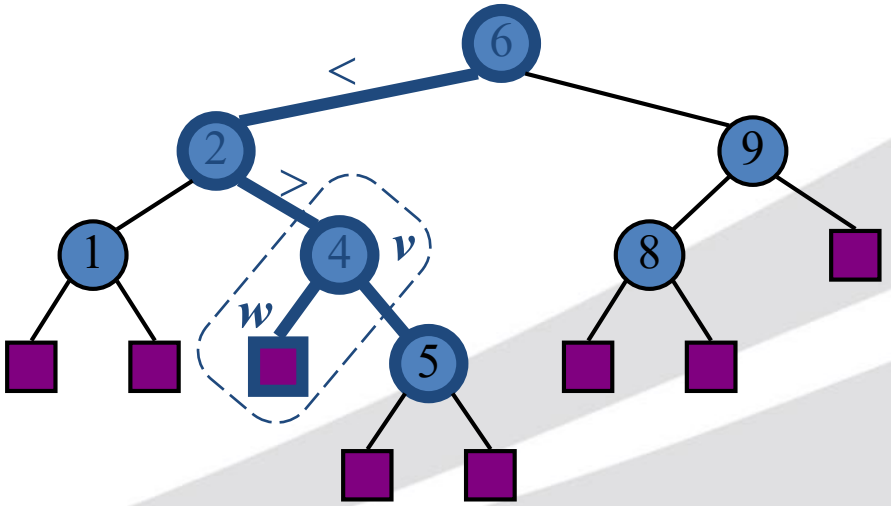
Ví dụ



Xóa nút trên cây

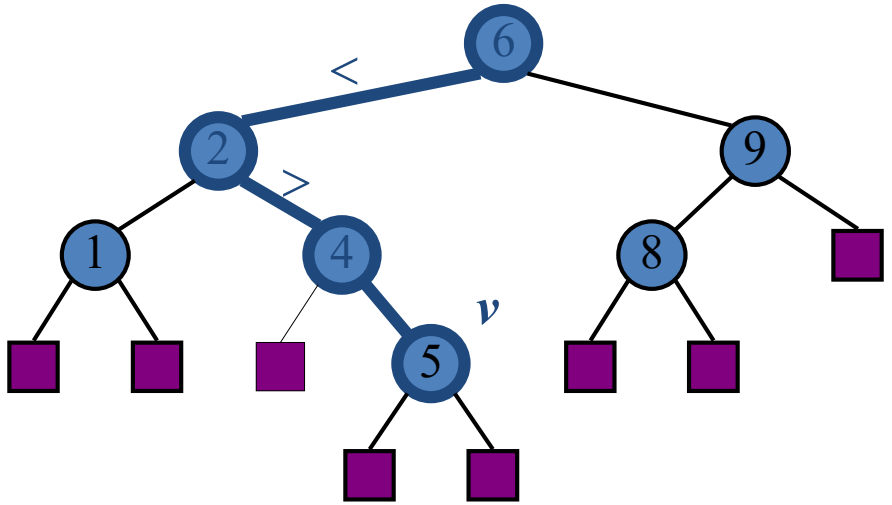
Sau khi bổ sung cây vẫn thỏa mãn tính chất cây TKNP

- **Xảy ra hai khả năng:**
  - Nút cần xóa là nút trong
  - Nút cần xóa là nút ngoài
- **Xóa một nút trong yêu cầu giải quyết lỗ hổng bên trong cây**
- **Để thực hiện thao tác remove(k), Chúng ta tìm kiếm đến nút có khóa k**
- **Giả sử rằng khóa k có ở trên cây, và ta đặt v là nút lưu trữ k**



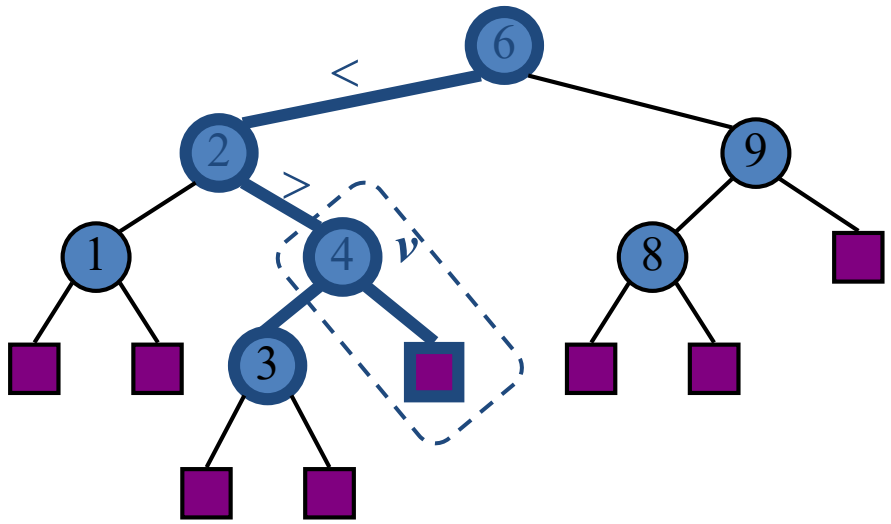
Xóa nút ngoài

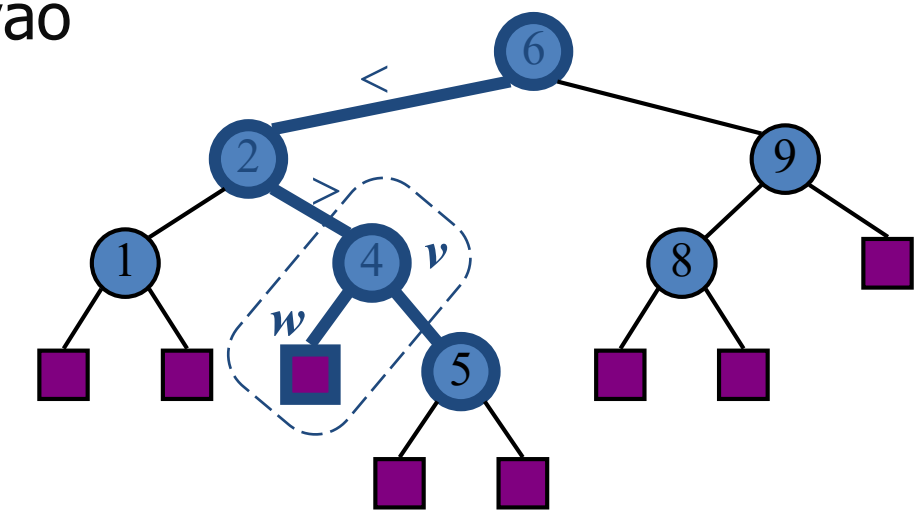
- Xóa bỏ nút *V*

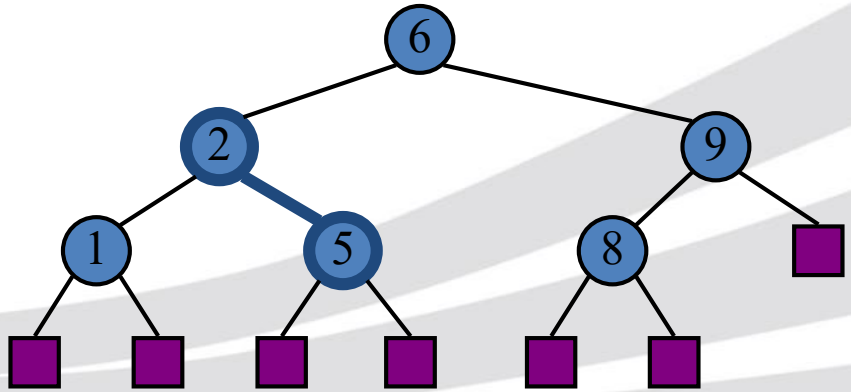


Xóa nút trong v chỉ có con trái hoặc phải

- Loại bỏ v và thay thế cây con của v vào vai trò của v
- Ví dụ: xóa bỏ 4

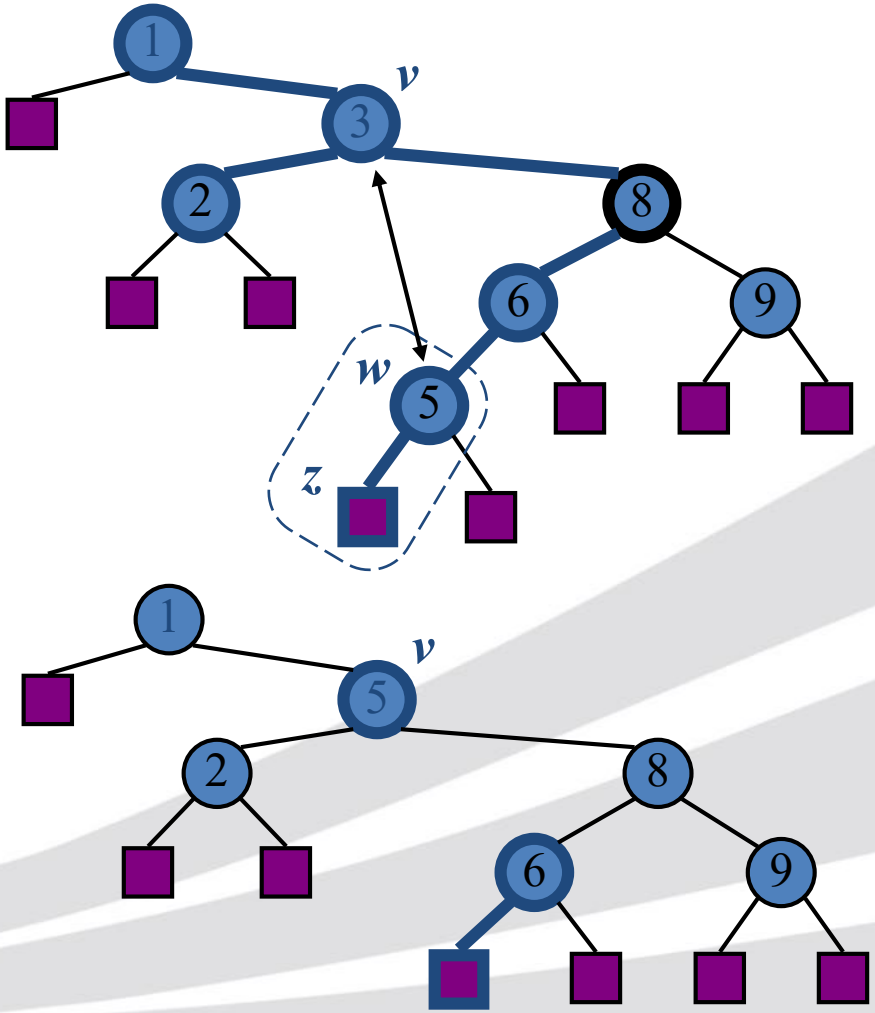






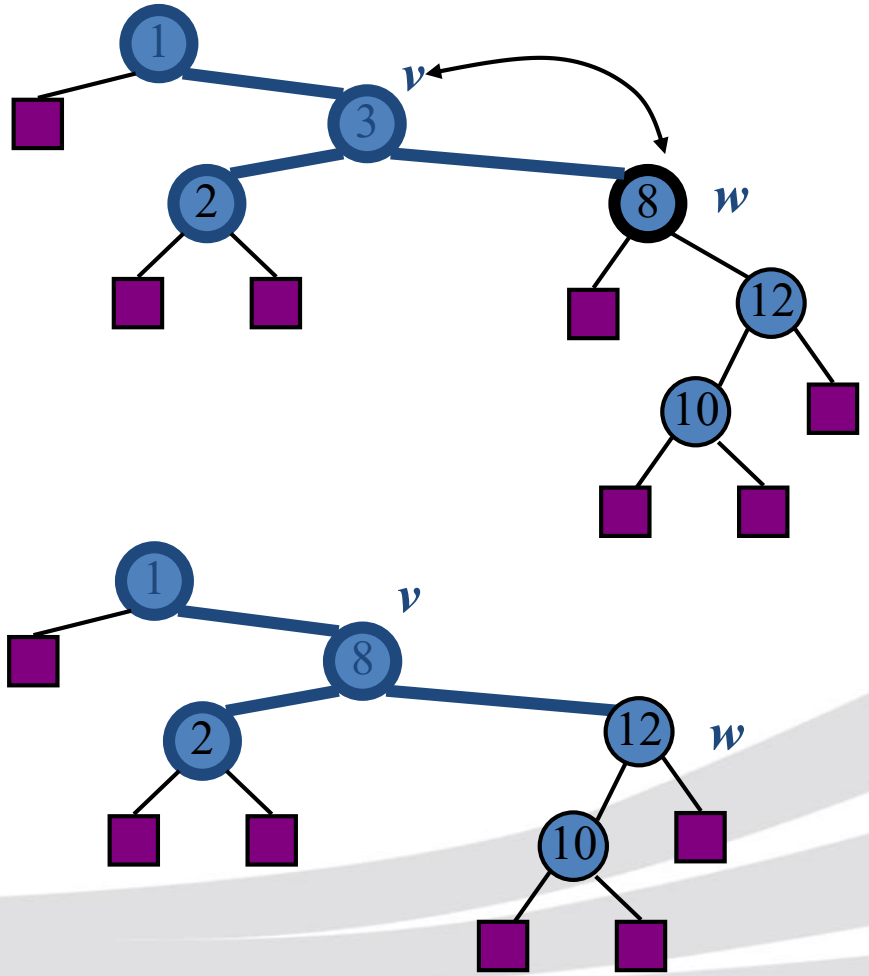
Xóa nút trong v có cả hai nút con trái và phải

- Xóa nút đó và thay thế nó bằng nút có khóa lớn nhất trong các khóa nhỏ hơn khóa của nó (được gọi là "nút tiền nhiệm" -nút cực phải của cây con trái) hoặc nút cón nhỏ nhất trong các khóa lớn hơn nó (được gọi là "nút kế vị" - nút cực trái của cây con phải) Cũng có thể tìm nút tiền nhiệm hoặc nút kế vị đổi chỗ nó với nút cần xóa và sau đó xóa nó. Vì các nút kiểu này có ít hơn hai con nên việc xóa nó được quy về hai trường hợp trước.
- Ví dụ: xóa bỏ 3



Ví dụ

- Xóa phần tử 3



Thuật toán giả mã

- Xây dựng hàm trả lại địa chỉ của nút nút bên trái nhất của cây con bên phải
- Xây dựng hàm xóa bỏ một nút trên cây

Hàm xóa bỏ một nút không có con hoặc chỉ có con trái hoặc con phải

```
void BTree<Keys,T>::remove(BNode<Keys,T> *v)
{
    BNode<Keys,T> *p;
    if (!v->hasLeft() && !v->hasRight())
    {
        p=v->getParent();
        if(p!=NULL)
           if(v == p->getLeft())
              p->setLeft(NULL);
           else
              p->setRight(NULL);
    }
    if(v->hasLeft() && !v->hasRight())
    {
        p=v->getParent();
        v->getLeft()->setParent(p);
        if(p->getLeft()==v)
            p->setLeft(v->getLeft());
        else
            p->setRight(v->getLeft());
    }
    if((!v->hasLeft()) && v->hasRight())
    {
        p=v->getParent();
        v->getRight()->setParent(p);
        if(p->getLeft()==v)
            p->setLeft(v->getRight());
        else
            p->setRight(v->getRight());
    }

    delete v;
}
```

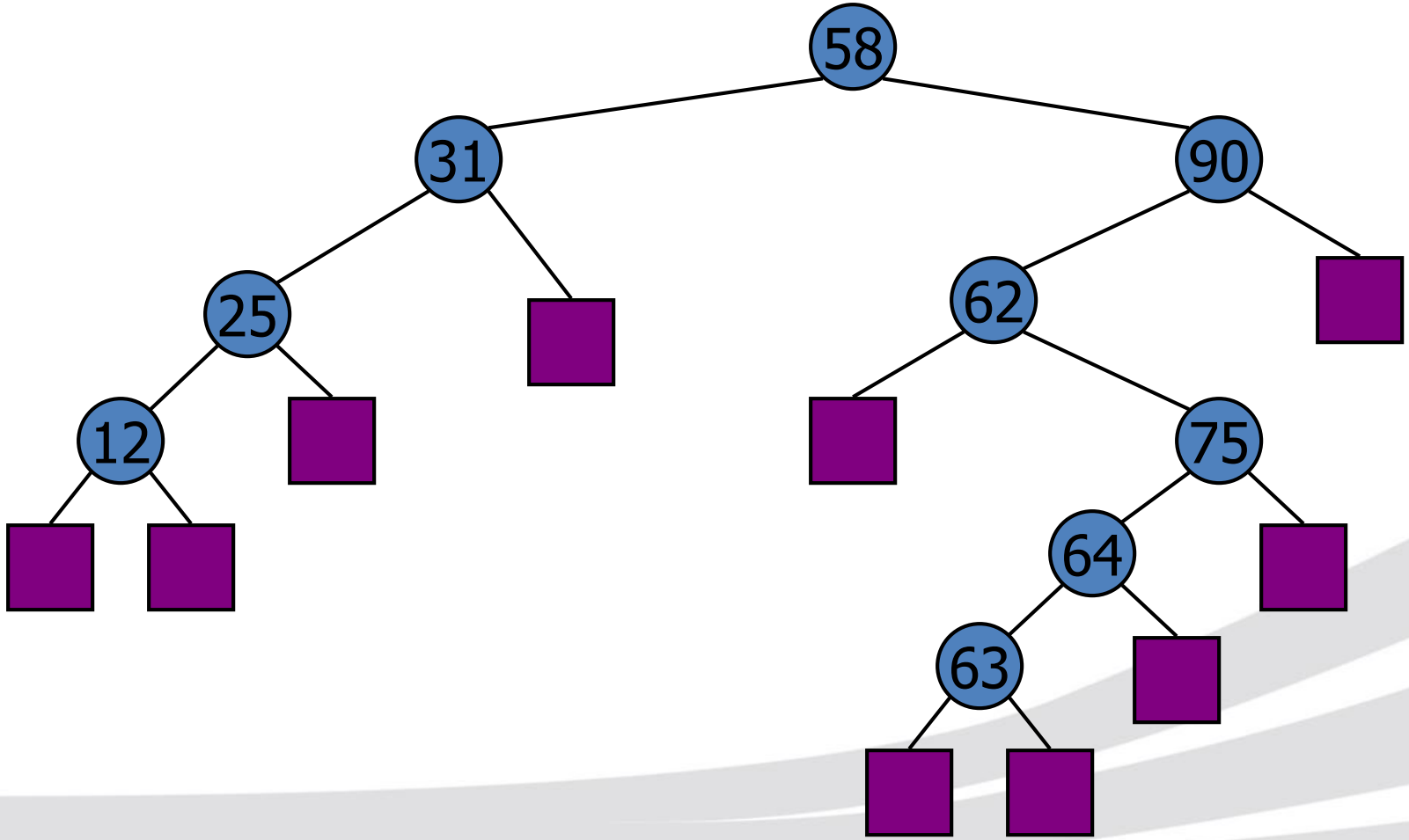
Hàm duyệt cây tìm nút con bên trái nhất

```
Node<Keys, T>* GetLeftest (Node<Keys, T>*v){
    Node<Keys, T> *p = v;
    while(p->getLeft()!=NULL)
        p=p->getLeft();
   return p;
}
```

Hàm xóa một nút bất kỳ

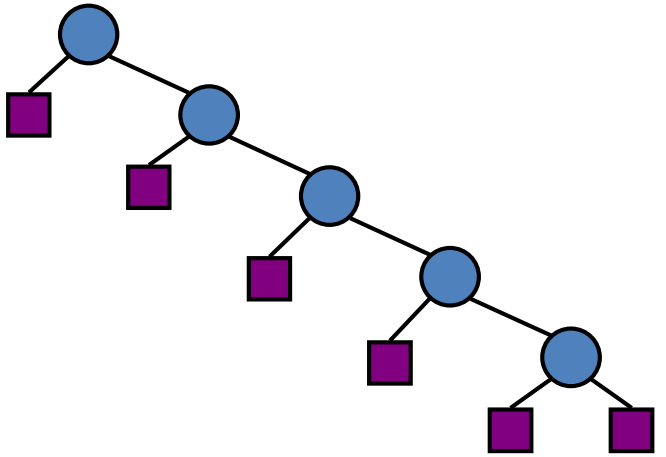
```
void  BTree<Keys,T>::remove(Keys key)
{
    BNode<Keys,T>*v = search(key, root);
    if(v==NULL) return;
    if(v->hasLeft()&& v->hasLeft())//Có ca hai con
    {
        BNode<Keys,T> *leftest;
        leftest = GetLeftest(v->getRight());
        v->setKey(leftest->getKey());
        v->setElem(leftest->getElem());
        remove(leftest);
    }
    else
        remove(v);
    count--;
}
```

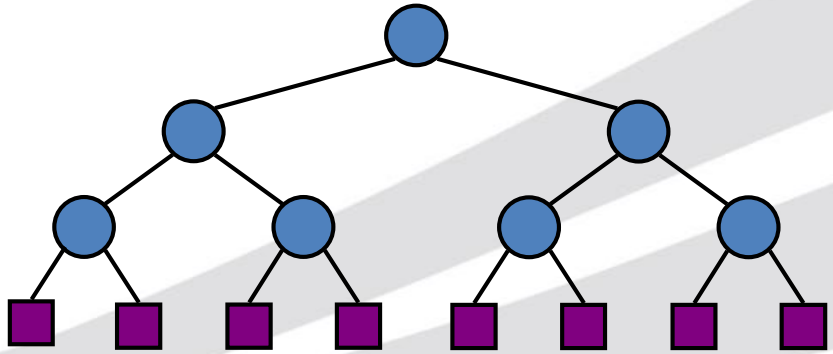
Ví dụ: Mô tả từng bước xóa bỏ nút có key=58?



Phân tích thời gian chạy

- **Xem xét việc cài đặt một từ điển có *n* mục được cài dặt bằng một cây nhị phân tìm kiếm với chiều cao *h***
  - Bộ nhớ sử dụng $O(n)$
  - Phương thức find, insert và remove take $O(h)$ time
- **Trong trường hợp xấu nhất chiều cao của cây là $O(n)$ và trường hợp tốt nhất là $O(\log n)$**





IV. Bài tập

1. **Xây dựng lớp cây tìm kiếm nhị phân**
2. **Sử dụng lớp cây tìm kiếm nhị phân xây dựng một chương trình tra cứu từ điển có các chức năng sau:**
   - Đọc dữ liệu từ điển nạp vào cây từ tệp
   - Bổ sung từ mới vào cây
   - Xóa bỏ một từ khỏi cây
   - Tìm kiếm từ
   - Lưu cây vào tệp

3. Hãy tạo cây TKNP từ dãy số sau: 20,12,43,5, 23, 78,53,9,4

Hết