Definição

Uma árvore é uma estrutura de dados composta por um conjunto de nós interconectados, que obedece às seguintes propriedades:

- **Nó raiz**: Existe um nó especial, chamado nó raiz, que é o ponto de partida da árvore.
- ´**Estrutura hierárquica**: A árvore possui uma organização hierárquica, onde cada nó pode ter um ou mais nós filhos, e cada nó, exceto a raiz, tem um nó pai.
- **Caminhos únicos**: Para quaisquer dois nós distintos *a* e *b*, existe, no máximo, um caminho único entre eles, o que garante que não há ciclos na árvore.

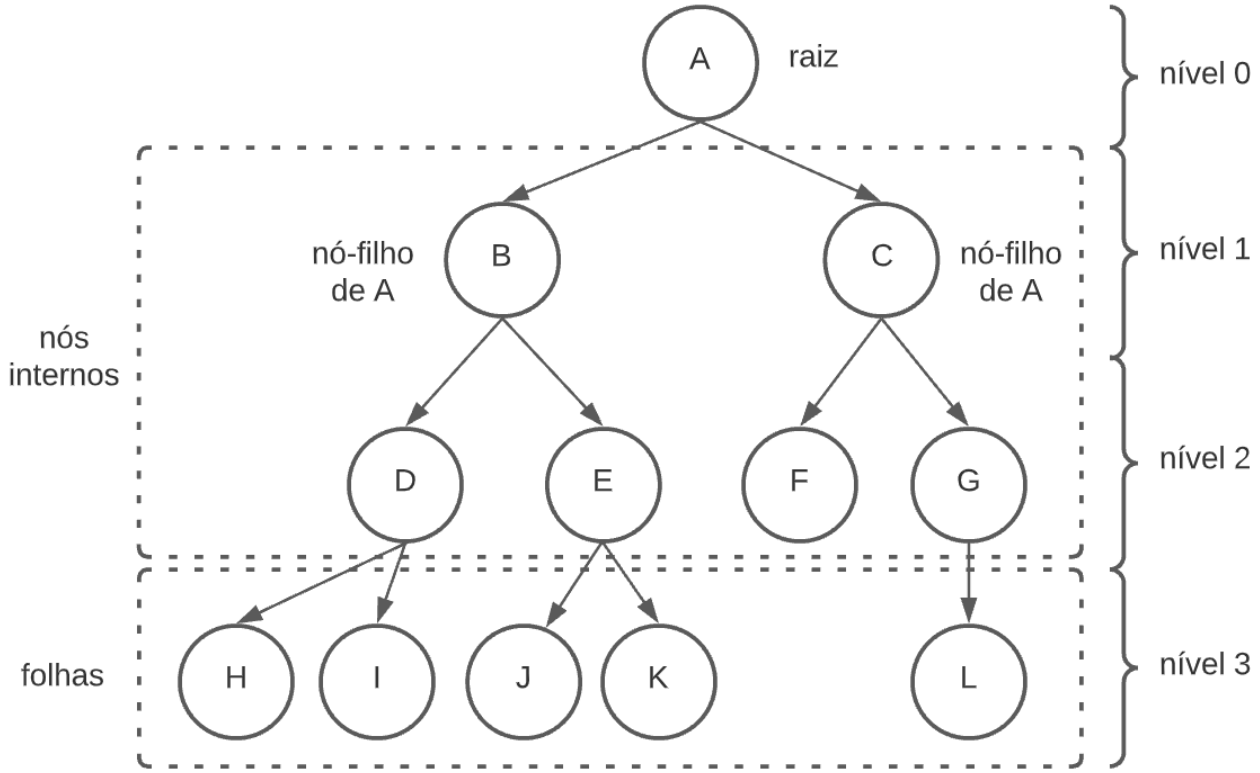


Figura 1: Representação de como funciona uma árvore, onde o nível 0 é o nó raiz, o nível 1 e 2 sendo os nós filhos, e o nível 3 sendo as folhas.

Árvore binária

Uma árvore binária é uma árvore em que cada nó tem no **MÁXIMO** 2 filhos (esquerdo e direito). Essa limitação torna as árvores binárias uma forma eficiente de armazenar e organizar dados.

Conceitos Fundamentais

- **Nó folha**: Um nó que não possui filhos é chamado de **folha** (Como representado no nível 3 da figura 1).
- **Descendentes**: Dado um nó *x*, qualquer nó que possa ser alcançado a partir de *x*, seja pela subárvore à esquerda ou à direita, é considerado um **descendente** de *x*.
- **Estrutura recursiva**: Uma árvore binária é, por natureza, uma estrutura recursiva. Isso significa que, dado um nó *x* qualquer, tanto o nó *x* quanto seus descendentes formam uma árvore binária. Se *x* for o filho esquerdo (ou direito) de um nó *y* dizemos que a árvore com raiz em *x* é uma subárvore esquerda (ou direita) de *y*.

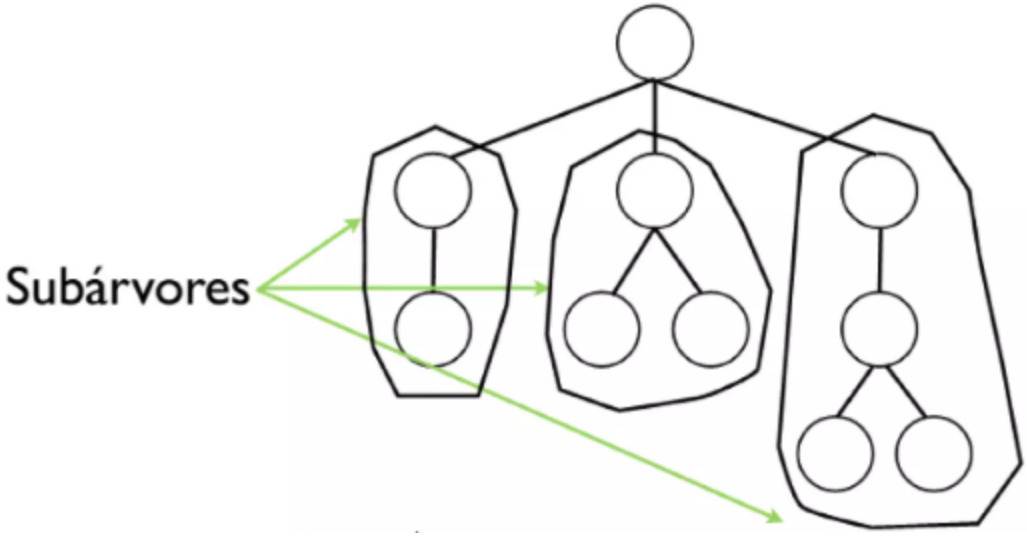


Figura 2: Demonstração da definição de subárvores.

- **Caminho e comprimento**: O caminho entre dois nós *x* e *y* é a sequência de nós percorridos para ir de um nó ao outro. O **comprimento do caminho** é o número total de nós atravessados.
- **Níveis e crescimento**: Os nós de uma árvore binária são organizados em níveis, numerados a partir de 0 (onde o nível 0 é a raiz, como mostrado na figura 1). O número máximo de nós em um nível *K* é 2^k, o que faz com que a árvore cresça exponencialmente à medida que avançamos para níveis mais profundos

---

Propriedades de Árvores Binárias

- **Árvore completa**: Uma árvore binária é chama de **completa** quando todos os seus níveis, do nível 0 até o nível *K*, estão completamente preenchidos, contendo exatamente 2^k nós em cada nível.
- **Altura de um nó**: A **altura** de um nó é definida como a distância (medida em número de arestas) entre ele e o seu descendente mais afastado, ou seja, o nó folha mais distante.
- **Profundidade de um nó**: A profundidade de um nó é a distância entre ele e a raiz da árvore, medida em número de arestas. A raiz, por definição, possui profundamente igual a

0.

- **Árvore balanceada**: uma árvore binária é considerada **balanceada** se, para cada nó, as alturas de suas subárvores esquerda e direita diferem no máximo por 1, ou, de forma equivalente, se todas as folhas possuem profundidades próximas. Árvores balanceadas são desejáveis porque garantem eficiência em operações como busca, inserção e remoção. Em geral, a altura de uma árvore binária balanceada com *n* nós é aproximadamente $\log_2 n$, tornando essas operações rápidas, com complexidade *O (log n)*.

Implementação

1. Estrutura do nó

```
typedef struct no{
        int dado; //Valor armazenado no nó
        struct no *esq, *dir; //Ponteiros para subárvore esquerda e direita
}no;
```

2. Função para criar um novo nó

```
No* criarNo(int valor){
        No* novo = (No*)malloc(sizeof(No));
        novo->dado = valor;
        novo->esq = NULL;
        novo->dir = NULL;
        return novo;
}
```

3. Função para inserir um valor na árvore

```
No* inserir(No* raiz, int valor){
        if(raiz == NULL) {
                return criarNo(valor); //caso base: insere o nó
        }
        if(valor < raiz->dado){
                raiz->esq = inserir(raiz->esq, valor); // Insere na subárvore
esquerda
        } else if (valor > raiz->dado){
                raiz->dir = inserir(raiz->dir, valor); // Insere na subárvore
direita
        }
        return raiz; // Retorna a raiz (inalterada)
}
```

4. Função para encontrar o menor valor em uma subárvore

```
No* menorValor(No* raiz) {
        No* atual = raiz;
        while (atual != NULL && atual->esq != NULL) {
                atual = atual->esq;
        }
        return atual;
}
```

5. Função para remover um nó da árvore

```
No* remover(No* raiz, int valor){
        if(raiz == NULL){
                return NULL; // Nó não encontrado
        }

        if(valor < raiz->dado){
                raiz->esq = remover(raiz->esq, valor); // Busca na subárvore
esquerda
        } else if(valor > raiz->dado){
                raiz->dir = remover(raiz->dir, valor); // Busca na subárvore
direita
        } else {
                // Caso 1: Nó folha
                if(raiz->esq == NULL && raiz->dir == NULL){
                        free(raiz);
                        return NULL;
                }
                //Caso 2: Um único filho
                else if(raiz->esq == NULL){
                        No* temp = raiz->dir;
                        free(raiz);
                        return temp;
                } else if(raiz->dir == NULL){
                        No* temp = raiz->esq;
                        free(raiz);
                        return temp;
                }
                // Caso 3: Dois filhos
                else {
                        No* temp = menorValor(raiz->dir); //menor valor na
subárvore direita
                        raiz->dado = temp->dado; // Substitui pelo valor do
```

```
sucessor
                        raiz->dir = remover(raiz->dir, raiz->dado); // Remove
o sucessor
                }
        }
        return raiz;
}
```

Percurso em profundidade na árvore binária

O percurso em profundidade explora os nós de uma árvore descendo o máximo possível em uma subárvore antes de retroceder. Ele pode ser realizado em três ordens principais:

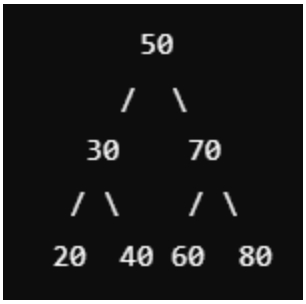
1. Pré-ordem
   Na pré-ordem, o nó atual é visitado antes de suas subárvores. A ordem é:

   1. Visite o nó atual
   2. Percorra a subárvore esquerda
   3. Percorra a subárvore direita

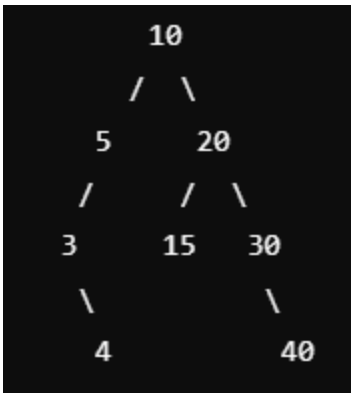
Aplicações

- Utilizado para copiar a estrutura da árvore
- Recomendado quando é necessário processar o nó antes de suas subárvores

Exemplos

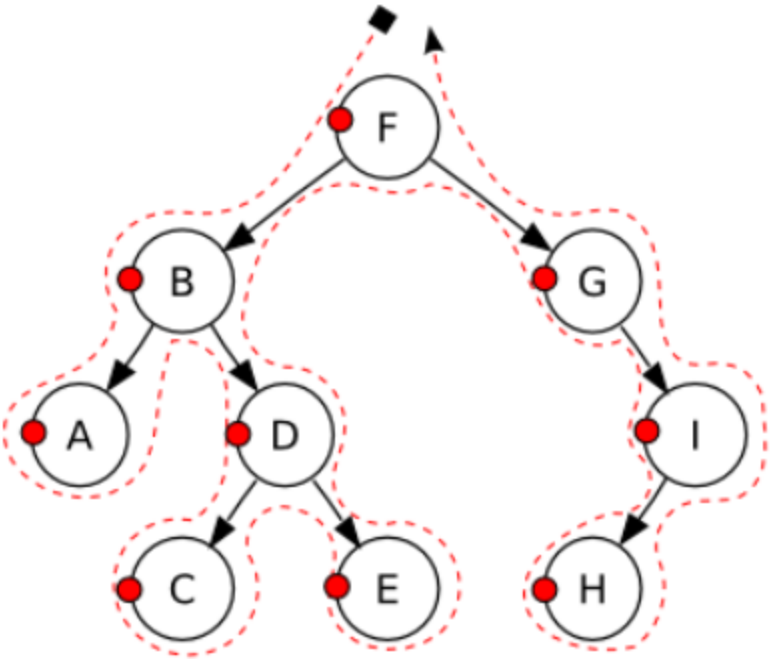


- **Sequência resultante**: 50, 30, 20, 40, 70, 60, 80.



- **Sequência resultante**: 10, 5, 3, 4, 20, 15, 30, 40.

Dica dos pontinho (à esquerda)



- **Sequência resultante**: F, B, A, D, C, G, I, H

Código

```c
void pre_ordem(No* raiz){
        printf("%d", raiz->dado); //Visita o nó atual
        pre_ordem(raiz->esq); // Percorre a subárvore esquerda
        pre_ordem(raiz->dir); // Percorre a subárvore direita
}
```

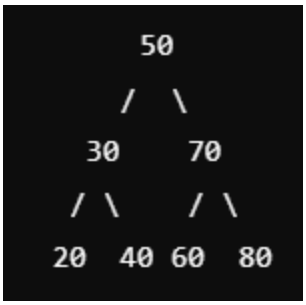
2. Pós-ordem
   Na pós-ordem, o nó atual é visitado após de suas subárvores. A ordem é:

1. Percorra a subárvore esquerda
2. Percorra a subárvore direita
3. Visite o nó atual

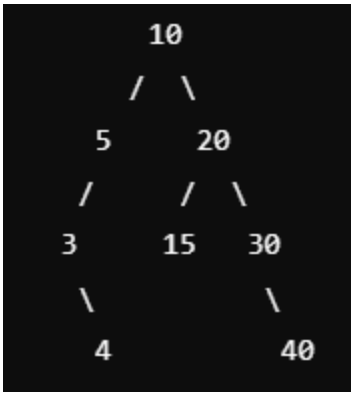
Aplicações

- Útil para apagar ou liberar memória da árvore
- Recomendado para realizar cálculos em estruturas hierárquicas

Exemplos

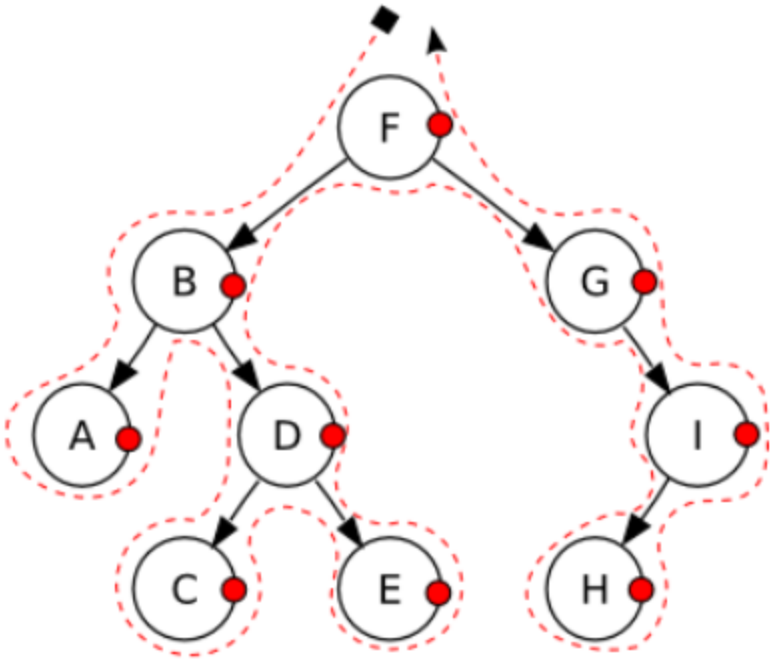


- **Sequência resultante**: 20, 40, 30, 60, 80, 70, 50.



- **Sequência resultante**: 4, 3, 5, 15, 40, 30, 20, 10.

Dica dos pontinho (à direita)



- **Sequência resultante**: A, C, E, D, B, H, I, G, F.

Código

```
void pos_ordem(No* raiz){
        pos_ordem(raiz->esq); // Percorre a subárvore esquerda
        pos_ordem(raiz->dir); // Percorre a subárvore direita
        printf("%d", raiz->dado); //Visita o nó atual
}
```

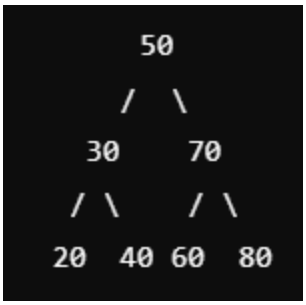
3. Em ordem
   Na em ordem, o nó atual é visitado entre as visitas às suas subárvores. A ordem é:

   1. Percorra a subárvore esquerda
   2. Visite o nó atual
   3. Percorra a subárvore direita

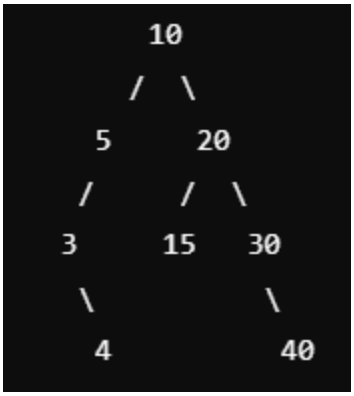
Aplicações

- Em uma árvore binária de busca (BST), o percurso em ordem resulta em uma lista ordenada
- Útil para exibir os valores em ordem crescente

Exemplos

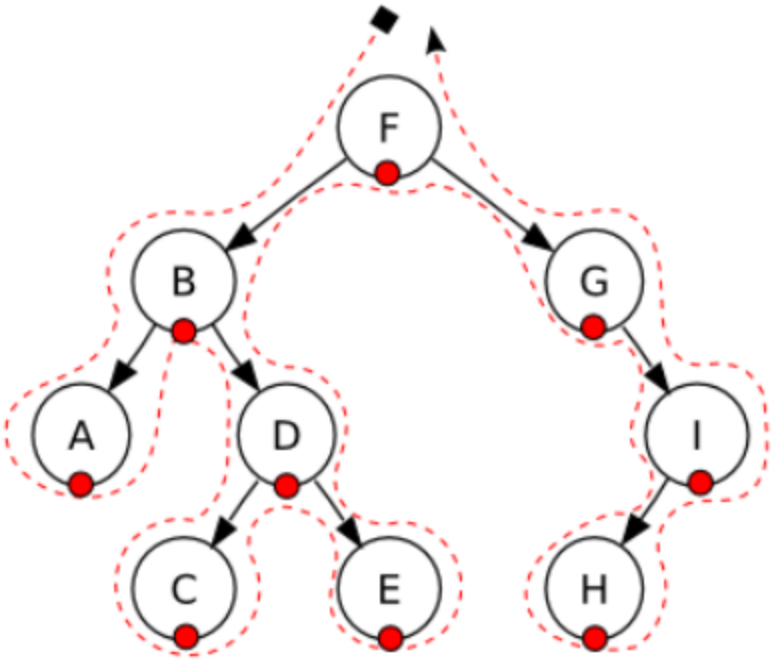


- **Sequência resultante**: 20, 30, 40, 50, 60, 70, 80.



- **Sequência resultante**: 4, 3, 5, 10, 15, 20, 30, 40.

Dica dos pontinhos (em baixo)



- **Sequência resultante**: A, B, C, D, E, F, G, H, I.

Código

```c
void em_ordem(No* raiz){
        em_ordem(raiz->esq); // Percorre a subárvore esquerda
        printf("%d", raiz->dado); //Visita o nó atual
        em_ordem(raiz->dir); // Percorre a subárvore direita
}
```

Árvore binária de busca (ABB)

É uma árvore binária que segue uma propriedade fundamental: para todo nó *x*, os valores armazenados nos nós de sua subárvore esquerda são menores ou iguais ao valor de *x*, enquanto os valores armazenados nos nós de sua subárvore direita são maiores que o valor de *x*. Formalmente, essa propriedade pode ser representada como:

$$e <= x <= d$$

Onde :

- *e* representa os valores nos nós da subárvore esquerda de x
- *d* representa os valores nos nós da subárvore direita de *x*

Propriedades

1. **Estrutura recursiva**: Cada subárvore de uma ABB também é uma ABB, pois ela respeita a mesma propriedade de ordenação
2. **Eficiência em operações**: A organização dos nós em uma ABB permite que operções como busca, inserção e remoção sejam realizadas de forma eficiente, com uma complexidade média de ***O(logn)*** em uma árvore balanceada
3. **Ordenação natural**: Um percurso em ordem em uma ABB sempre gera os elementos em ordem crescente

Operações

1. Busca :
   A busca na ABB aproveita sua estrutura ordenada:
   - Começa na raiz
   - Se o valor procurado for menor que o valor do nó atual, segue para subárvore esquerda
   - Se for maior, segue para a subárvore direita
   - Se encontrar o valor, a busca termina

Complexidade:
- Melhor caso: $O(logn)$
- Pior caso: $O(n)$ -> árvore desbalanceada

Código:

```
        No* bucar(No* raiz, int valor){
                if(raiz == NULL || raiz->dado == valor){
                        return raiz; // Retorna o nó encontrado ou NULL se não
existir
                }
                if(valor < raiz ->dado){
                        return buscar(raiz->esq, valor);
                }else{
                        return buscar(raiz->dir, valor);
                }

        }
```

2. Inserção :
   Para inserir um elemento em uma ABB:
   - Começa na raiz e desce até encontrar um nó folha
   - Compara o valor a ser inserido com o valor dos nós ao longo do caminho:
   - Se for menor, move-se para a esquerda
   - Se for maior, move-se para a direita
   - Insere o novo nó no local adequando, mantendo a propriedade da ABB
   Código:

```
        No* inserir(No* raiz, int valor){
                if(raiz == Null){
                        return criarNo(valor);
                }
                if(valor < raiz->dado){
                        raiz->esq = inserir(raiz->esq, valor);
                } else if( valor > raiz->dado){
                        raiz->dir = inserir(raiz->dir, valor);
                }
                return raiz;
        }
```

3. Remoção:
   A remoção é um pouco mais complexa e possui três casos principais:
   - Nó é uma folha: remove-se diretamente

- Nó tem um único filho: substitui-se o nó pelo seu filho
- Nó tem dois filhos: substitui-se o nó pelo menor valor da sua subárvore direita (o sucessor) ou pelo maior valor da subárvore esquerda (o predecessor) e remove-se o nó substituído.

Código:

```
//Função para encontrar o menor valor em uma subárvore
No* menorValor(No* raiz){
        No* atual = raiz;
        while(atual && atual->esq != NULL){
                atual = atual->esq;
        }
        return atual
}


//Função para remover nó
No* remover(No* raiz, int valor){
        if(raiz == NULL){
                return raiz; // Caso base: árvore vazia
        }

        if(valor < raiz->dado){
                raiz->esq = remover(raiz->esq, valor);// Busca na subárvore
esquerda
        }else if(valor > raiz->dado){
                raiz->dir = remover(raiz->dir, valor);// Busca na subárvore
direita
        } else {

                //Caso 1
                if(raiz->esq == NULL && raiz->dir == NULL){
                        free(raiz);
                        return NULL;
                }

                //Caso 2
                if(raiz->esq == NULL){
                        No* temp = raiz->dir;
                        free(raiz);
                        return temp;
                }else if(raiz->dir == NULL){
                        No* temp = raiz->esq;
                        free(raiz);
                        return temp;
```

```
                }


                //Caso 3
                No* temp = menorValor(raiz->dir); // Encontra o sucessor in-
order
                raiz->dado = temp->dado; // Substitui o valor do nó pelo
sucessor
                raiz->dir = remover(raiz->dir, temp->dado); // Remove o
sucessor

        }

        return raiz;

}
```

ABB X BB

- **Dados Estáticos e Ordenados: Busca Binária**
  - **Eficiência em Pesquisas:** A busca binária é extremamente eficiente em listas ordenadas, reduzindo o espaço a procurar pela metade a cada iteração. Ela opera com complexidade de tempo $O(\log n)$.
  - **Simples Implementação:** Implementar uma busca binária é direto, exigindo uma estrutura simples de vetor ou lista.
  - **Memória Estática:** Como os dados são estáticos, não há necessidade de memória adicional para gerenciamento dinâmico, tornando-a mais leve em uso de memória.
  - **Limitação em Dados Dinâmicos:** A busca binária não se adapta bem a dados dinâmicos porque qualquer modificação no conjunto de dados requer manter a ordenação, muitas vezes exigindo reprocessamento dispendioso.
- **Dados Dinâmicos e Desordenados: Árvore de Busca Binária (ABB)**
  - **Flexibilidade em Modificações:** ABBs são ideais para conjuntos de dados dinâmicos, permitindo inserções, remoções e pesquisas eficientes com complexidade média de $O(\log n)$ quando balanceadas.
  - **Autobalanceamento Possível:** Com implementações como Árvores AVL ou Árvores Rubro-Negras, a ABB pode ser mantida equilibrada automaticamente, melhorando o desempenho.
  - **Gerenciamento de Ordenação Interna:** ABBs fornecem manutenção automática da ordenação dos dados, eliminando a necessidade de reordenamentos frequentes.
  - **Overhead de Estrutura:** ABBs requerem mais memória e complexidade estrutural comparado a uma lista simples, devido aos apontadores e operações de

balanceamento.
- **Processamento de Dados Desordenados:** Por construírem sua estrutura à medida que os dados chegam, são muito úteis quando não se pode garantir um conjunto ordenado de entrada a priori.

Esses argumentos demonstram que a escolha entre busca binária e ABB depende fortemente da natureza dos dados e dos requisitos de operação. Em cenários do mundo real, fatores como a frequência de atualização dos dados e a necessidade de ordenação dinamicamentestambém devem ser considerados na escolha do método mais adequado.

Antecessor e Sucessor

Antecessor de um nó *x* é o maior elemento na subárvore esquerda de *x*. Em termos práticos:

- A subárvore direita do antecessor é vazia, pois o antecessor está na maior posição possível à esquerda.

Por outro lado, o sucessor do nó *x* é o menor elemento da subárvore direita de *x*. Isso implica que:

- A subárvore esquerda do sucessor é vazia, pois o sucessor ocupa a menor posição possível à direita.

Condições especiais

- **Caso a subárvore esquerda de *x* seja vazia**:
  - O antecessor será o primeiro ancestral encontrado à esquerda durante a subida na árvore
- **Caso a subárvore direita de *x* seja vazia**:
  - O sucessor será o primeiro ancestral encontrado à direita durante a subida na árvore

Complexidade de busca

- Pior caso : $O(n)$
- Melhor caso : $O(logn)$

Exemplo

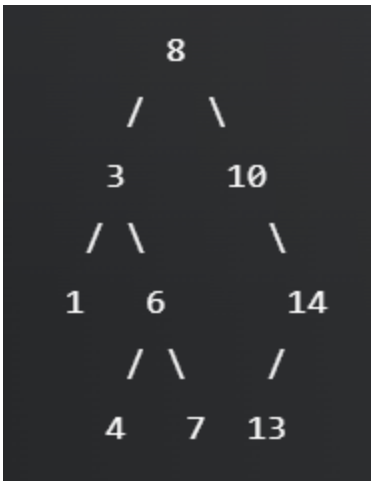


Figura 3: Exemplo de árvore binárias

Vamos determinar o antecessor e o sucessor para os principais nós nesta árvore:

1. **Nó 1:**
   - **Antecessor:** Não tem (é o menor elemento da árvore)
   - **Sucessor:** 3
2. **Nó 3:**
   - **Antecessor:** 1
   - **Sucessor:** 4 (menor elemento na subárvore direita)
3. **Nó 4:**
   - **Antecessor:** 3
   - **Sucessor:** 6
4. **Nó 6:**
   - **Antecessor:** 4
   - **Sucessor:** 7
5. **Nó 7:**
   - **Antecessor:** 6
   - **Sucessor:** 8
6. **Nó 8:**
   - **Antecessor:** 7
   - **Sucessor:** 10
7. **Nó 10:**
   - **Antecessor:** 8
   - **Sucessor:** 13
8. **Nó 13:**
   - **Antecessor:** 10
   - **Sucessor:** 14

9. **Nó 14:**
    - **Antecessor:** 13
    - **Sucessor:** Não tem (é o maior elemento da árvore)

Isto ilustra como o antecessor e o sucessor de cada nó são determinados analisando as propriedades das subárvores. Estas propriedades são úteis em operações de inserção, exclusão, e balanceamento de árvores binárias de busca.

Código

1. Antecessor

```
// Função para encontrar o maior nó na subárvore
No* maior(No* x) {
    while (x->dir != NULL) {
        x = x->dir;
    }
    return x;
}

// Função para encontrar o ancestral à esquerda
No* ancestral_a_esq(No* x, No* raiz) {
    No* anc = NULL;
    while (x != raiz) {
        if (x->dado > raiz->dado) {
            // Se o dado do nó `x` for maior, vá para a direita
            raiz = raiz->dir;
        } else {
            // Talvez `raiz` seja um possível ancestral à esquerda
            anc = raiz;
            raiz = raiz->esq;
        }
    }
    return anc;
}

// Função para encontrar o antecessor
No* antecessor(No* x, No* raiz) {
    if (x->esq != NULL) {
        // Se existe um subárvore à esquerda, o antecessor está nela
        return maior(x->esq);
    } else {
        // Caso contrário, buscamos entre os ancestrais
        return ancestral_a_esq(x, raiz);
```

```
    }
}
```

- **Função `maior`:** Esta função é responsável por encontrar o maior elemento em uma subárvore. Ela percorre a subárvore à direita até encontrar o nó mais à direita.
- **Função `ancestral_a_esq`:** Esta função busca o primeiro ancestral no caminho à esquerda. Isso é útil quando a subárvore esquerda de um nó é nula, e precisamos procurar o antecessor entre os ancestrais.
- **Função `antecessor`:** Ela verifica se o nó tem uma subárvore esquerda. Se sim, usa a função `maior` para encontrar o antecessor. Caso contrário, delega para `ancestral_a_esq` para encontrar o antecessor entre os ancestrais.

2. Sucessor

```
// Função para encontrar o menor nó na subárvore
No* menor(No* x) {
    while (x->esq != NULL) {
        x = x->esq;
    }
    return x;
}

// Função para encontrar o ancestral à direita
No* ancestral_a_direita(No* x, No* raiz) {
    No* anc = NULL;
    while (x != raiz) {
        if (x->dado < raiz->dado) {
            // Se o dado de `x` for menor, `raiz` é um possível ancestral à
direita
            anc = raiz;
            raiz = raiz->esq;
        } else {
            // Vá para a direita
            raiz = raiz->dir;
        }
    }
    return anc;
}

// Função para encontrar o sucessor
No* sucessor(No* x, No* raiz) {
    if (x->dir != NULL) {
        // Se existe uma subárvore à direita, o sucessor está nela
        return menor(x->dir);
```

```
    } else {
        // Caso contrário, buscamos entre os ancestrais
        return ancestral_a_direita(x, raiz);
    }
}
```

- **Função `menor`:** Esta função encontra o menor elemento em uma subárvore, e é análoga à função `maior`, usada na busca do antecessor.
- **Função `ancestral_a_direita`:** Serve para encontrar o primeiro ancestral no caminho à direita, utilizado quando a subárvore direita de um nó é nula, indicando que precisamos buscar o sucessor entre seus ancestrais.
- **Função `sucessor`:** Verifica se o nó tem uma subárvore direita. Se sim, usa a função `menor` para encontrar o sucessor. Caso contrário, usa `ancestral_a_direita` para procurar o sucessor entre os ancestrais.