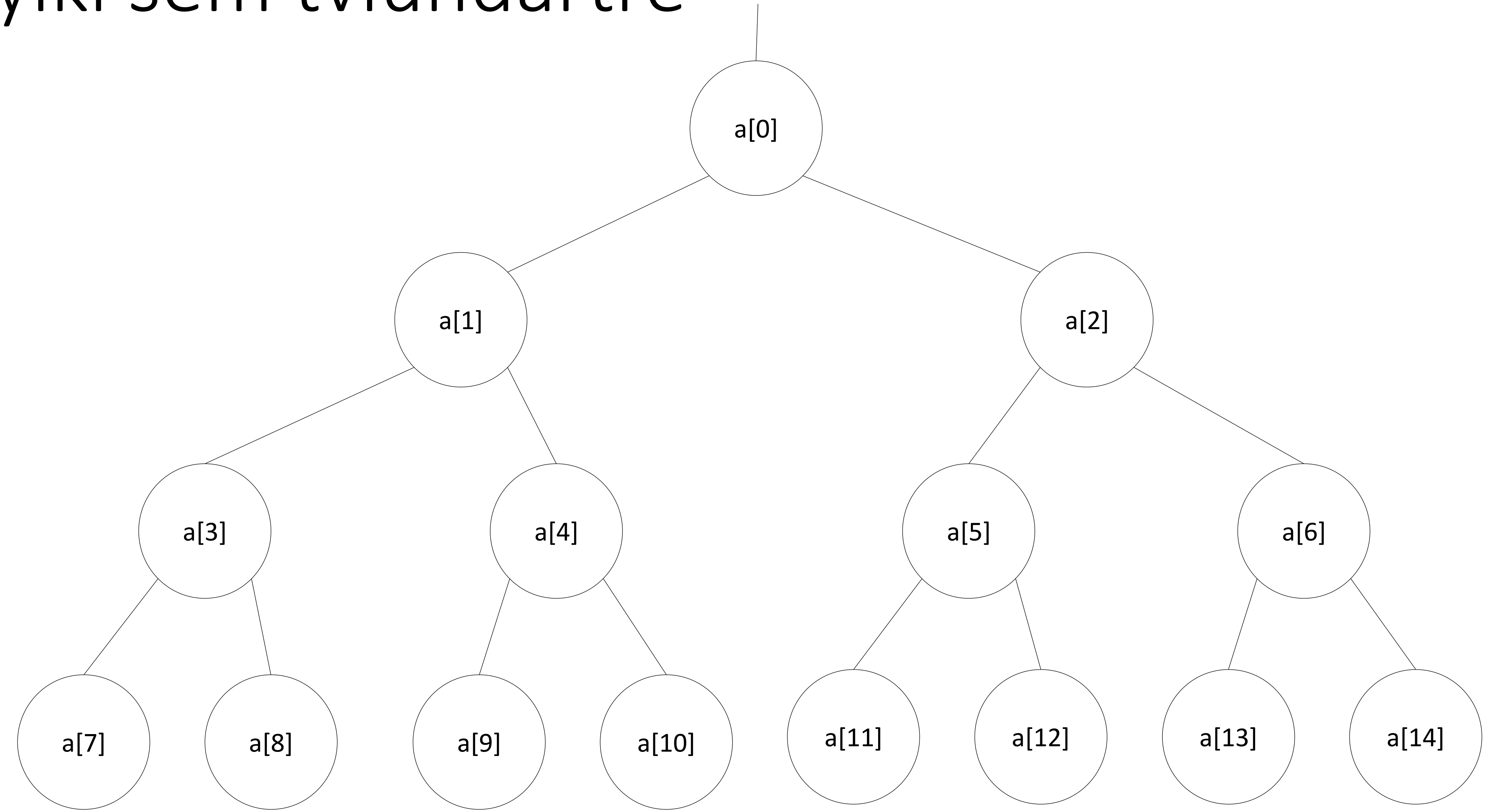
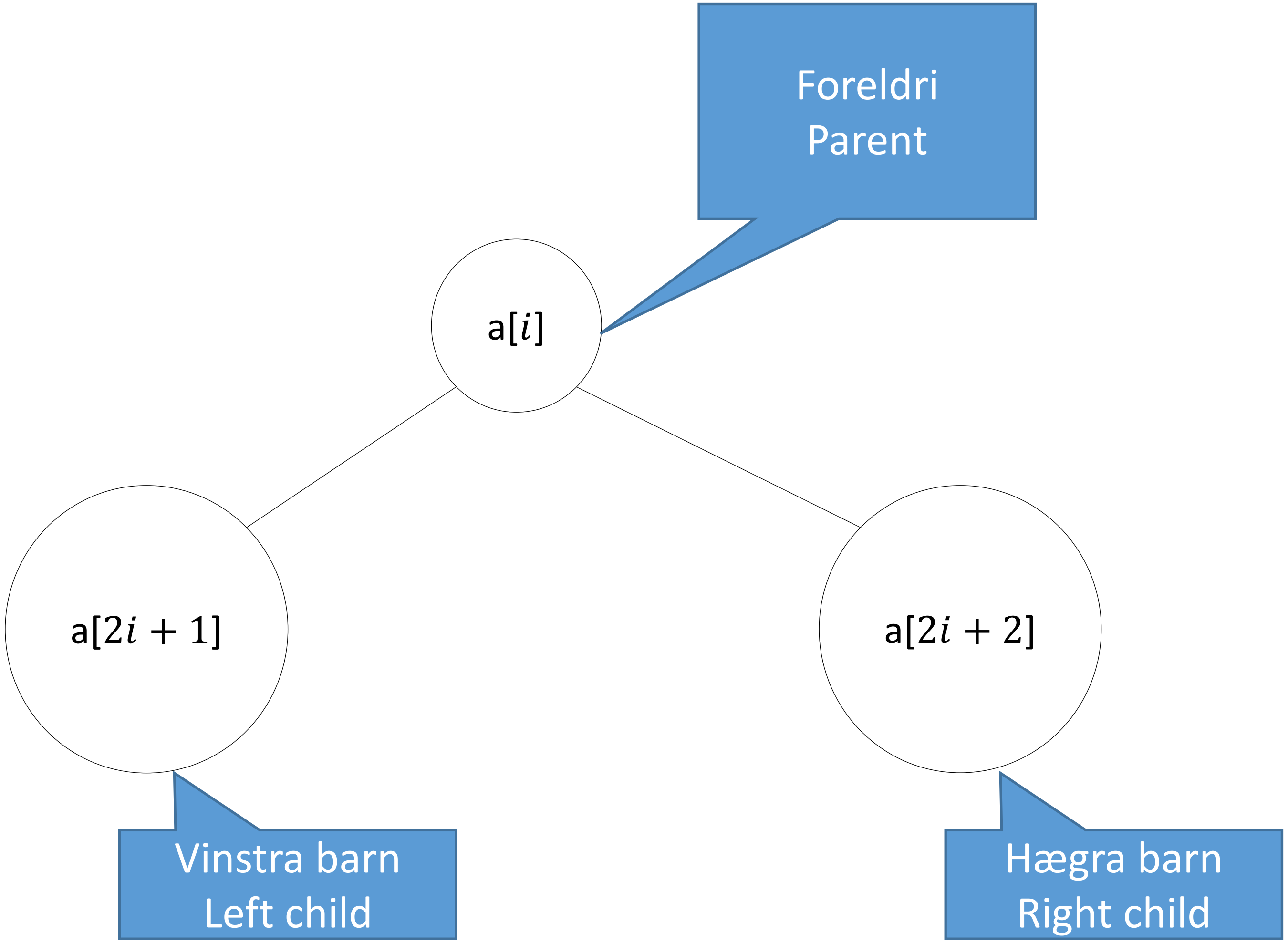
Hrúgur – Heaps

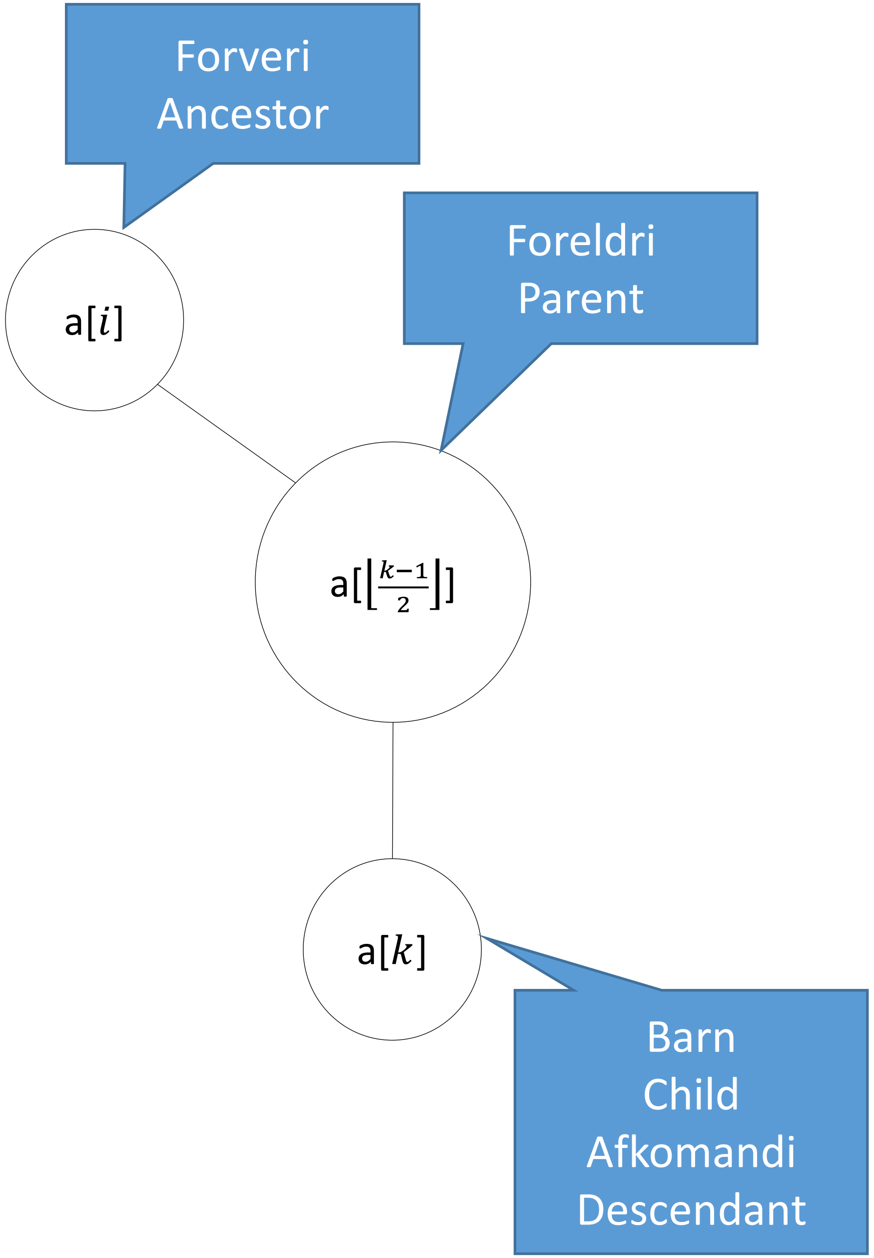
Sjá einnig Heap.dfy í Canvas

Fylki sem tvíundartré



Foreldri, börn, forverar og afkomendur





Hrúga – Heap

- Fylki a **er minhrúga** (minheap) þá og því aðeins að fyrir sérhverja tvo vísa p og q innan fylkisins þar sem a$[p]$ er forveri a$[q]$ gildir a$[p]$≤a$[q]$
- Fylki a **er maxhrúga** (maxheap) þá og því aðeins að fyrir sérhverja tvo vísa p og q innan fylkisins þar sem a$[p]$ er forveri a$[q]$ gildir a$[p]$≥a$[q]$
- Ef a er minhrúga þá er minnsta gildið í sæti a[0]
- Ef a er maxhrúga þá er stærsta gildið í sæti a[0]
- Svæði innan fylkis a uppfyllir **minhrúguskilyrði** þá og því aðeins að fyrir sérhverja tvo vísa p og q innan svæðisins þar sem a$[p]$ er forveri a$[q]$ gildir a$[p]$≤a$[q]$
- Svæði innan fylkis a uppfyllir **maxhrúguskilyrði** þá og því aðeins að fyrir sérhverja tvo vísa p og q innan svæðisins þar sem a$[p]$ er forveri a$[q]$ gildir a$[p]$≥a$[q]$

Dafny umsögn

```
predicate IsAncestorOf( p: int, q: int )
    decreases q;
    requires 0 <= p;
    requires 0 <= q;
{
     p < q && (p == (q-1)/2 || IsAncestorOf(p,(q-1)/2))
}
```

Dafny umsögn

```
predicate IsMinHeap( a: seq<int>, i: int, j: int )
    requires 0 <= i <= j <= |a|;
{
    forall p,q | i <= p < q < j && IsAncestorOf(p,q) ::
        a[p] <= a[q]
}
```

Röðun og forgangsbiðraðir með hrúgum

- Auðvelt er að nota hrúgur til að raða (heapsort)
- Algengasta aðferðin er að nota hjálparfall sem stækkar svæði sem uppfyllir hrúguskilyrði, **rolldown** fall
  - Fallið **rolldown** víxlar gildi niður tréð uns það er komið á réttan stað
- Einnig má nota tvö mismunandi hjálparföll, þ.e. **rolldown** fall og einnig **rollup**, sem stækkar svæðið í hina áttina
  - Fallið **rollup** víxlar gildi upp tréð (í átt að rótinni) uns það er rétt
- Þessi tvö hjálparföll má einnig nota til að útfæra forgangsbiðraðir (priority queue) með hrúgum

Dafny umsögn

```
predicate IsMinHeapRollingDown
     ( a: seq<int>, i: int
     , k: int, j: int
     )
   requires 0 <= i <= k < j <= |a|;
{
   forall p,q |    i <= p < q < j &&
                   IsAncestorOf(p,q) &&
                   p != k ::
                       a[p] <= a[q]
}
```

Þessi umsögn er gagnleg sem stöðulýsing þegar verið er að rúlla gildi niður tréð.
Hún er sönn ef svæðið a[i..j] uppfyllir hrúguskilyrði fyrir utan að gildið í sæti k er **e.t.v. of ofarlega** í trénu.

Dafny umsögn

```
predicate IsMinHeapRollingUp
     ( a: seq<int>, i: int
     , k: int, j: int
     )
    requires 0 <= i <= k < j <= |a|;
{
    forall p,q |    i <= p < q < j &&
                    IsAncestorOf(p,q) &&
                    q != k ::
                        a[p] <= a[q]
}
```

Þessi umsögn er gagnleg sem stöðulýsing þegar verið er að rúlla gildi upp tréð.
Hún er sönn ef svæðið a[i..j] uppfyllir hrúguskilyrði fyrir utan að gildið í sæti k er **e.t.v. of neðarlega** í trénu.

Dafny hjálparsetning

```
lemma TransitiveAncestor( p: int, q: int, r: int )
    decreases r;
    requires 0 <= p;
    requires 0 <= q;
    requires 0 <= r;
    requires IsAncestorOf(p,q);
    requires IsAncestorOf(q,r);
    ensures  IsAncestorOf(p,r);
{
    if q == (r-1)/2 { return; }
}
```

Dafny hjálparsetning

```
lemma DescendantIsChildOrDescendantOfChild( p: int, q: int )
    requires 0 <= p;
    requires 0 <= q;
    requires IsAncestorOf(p,q);
    ensures q==2*p+1 ||
            q==2*p+2 ||
            IsAncestorOf(2*p+1,q) ||
            IsAncestorOf(2*p+2,q);
{}
```

Dafny hjálparsetning

```
lemma AllDescendantsAreChildrenOrDescendantsOfChild( p: int )
    requires 0 <= p;
    ensures forall q | q >=0 && IsAncestorOf(p,q) ::
        q==2*p+1 ||
        q==2*p+2 ||
        IsAncestorOf(2*p+1,q) ||
        IsAncestorOf(2*p+2,q);
{
    forall q | q>=0 && IsAncestorOf(p,q)
    {
        DescendantIsChildOrDescendantOfChild(p,q);
    }
}
```

Dafny hjálparsetning

```
lemma MinHeapRollingDownBecomesHeap( a: seq<int>, i: int, k: int, j: int )
    requires 0 <= i <= k < j <= |a|;
    requires IsMinHeapRollingDown(a,i,k,j);
    ensures 2*k+1 >= j ==> IsMinHeap(a,i,j);
    ensures 2*k+1 < j && 2*k+2 >= j && a[k] <= a[2*k+1] ==> IsMinHeap(a,i,j);
    ensures 2*k+2 < j && a[k] <= a[2*k+1] && a[k] <= a[2*k+2] ==> IsMinHeap(a,i,j);
{
    if 2*k+1 >= j { return; }
    if 2*k+1 < j && 2*k+2 >= j && a[k] <= a[2*k+1] { return; }
    if 2*k+2 < j && a[k] <= a[2*k+1] && a[k] <= a[2*k+2]
    {
        if IsMinHeap(a,i,j) { return; }
        // Gefið að IsMinHeap(a,i,j) er ósatt munum við leiða út mótsögn
        var p,q :| i <= p < q < j && IsAncestorOf(p,q) && a[p] > a[q];
        DescendantIsChildOrDescendantOfChild(p,q);
        assert false; // Já! Við höfum mótsögn, svo IsMinHeap(a,i,j) er satt
    }
}
```

Dafny hjálparsetning

```
lemma ZeroIsRoot( p: int )
    decreases p;
    requires p > 0;
    ensures IsAncestorOf(0,p);
{
    if p == 1 || p == 2 { return; }
    ZeroIsRoot((p-1)/2);
    TransitiveAncestor(0,(p-1)/2,p);
}
```

Dafny hjálparsetning

```
lemma RootHasMin( a: seq<int>, i: int, j: int )
    requires 0 <= i <= j <= |a|;
    requires IsMinHeap(a,i,j);
    ensures forall p | i <= p < j && IsAncestorOf(i,p) :: a[i] <= a[p];
{}
```

Dafny hjálparsetning

```
lemma ZeroHasMin( a: seq<int>, n: int )
    requires 0 < n <= |a|;
    requires IsMinHeap(a,0,n);
    ensures forall p | 0 < p < n :: IsAncestorOf(0,p);
    ensures forall p | 0 <= p < n :: a[0] <= a[p];
{
    forall p | 1 <= p < n { ZeroIsRoot(p); }
}
```

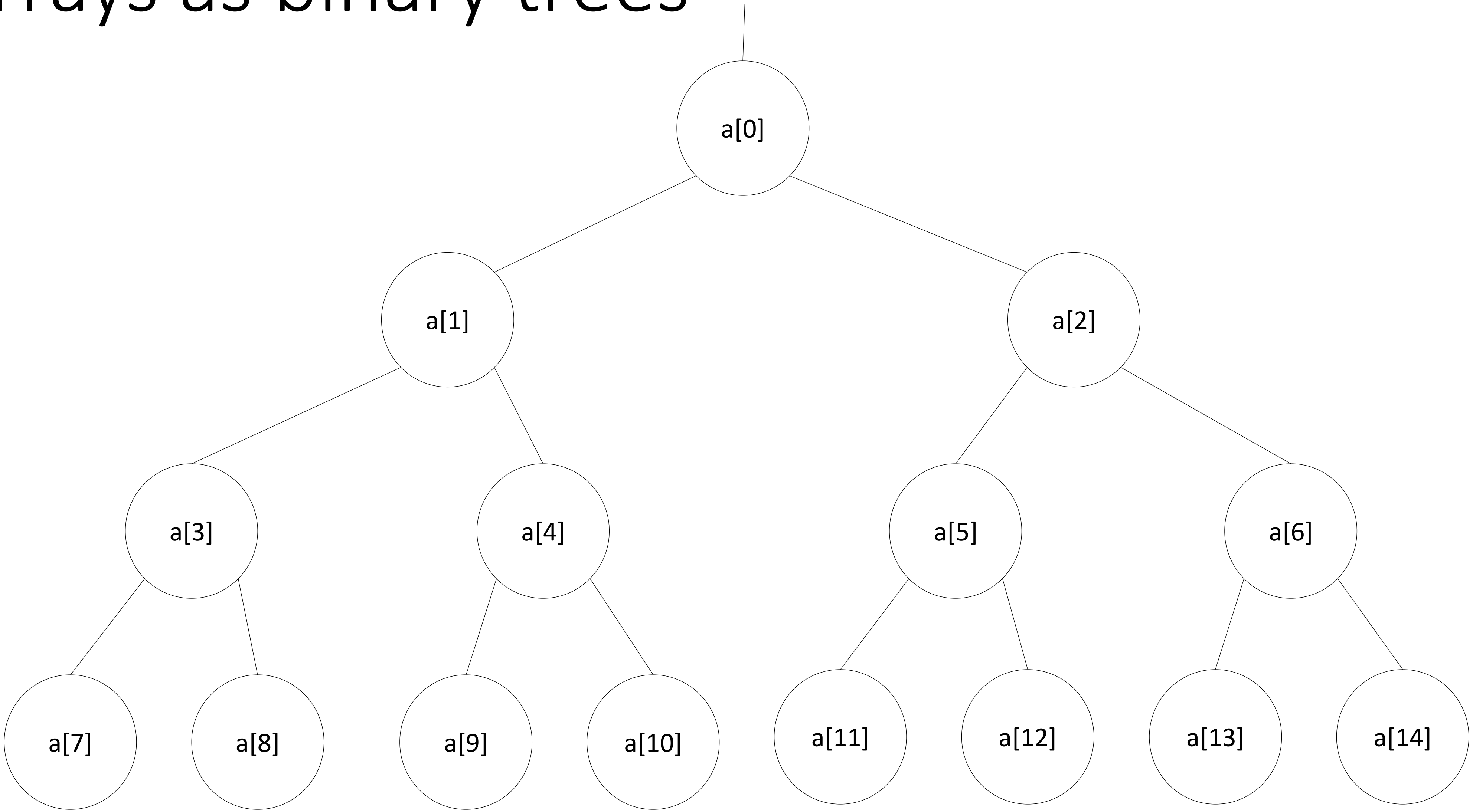
Dafny hjálparsetning

```
lemma PartOfHeap( a: seq<int>, i: int, j: int, p: int, q: int )
    requires 0 <= i <= p <= q <= j <= |a|;
    requires IsMinHeap(a,i,j);
    ensures IsMinHeap(a,p,q);
{}
```

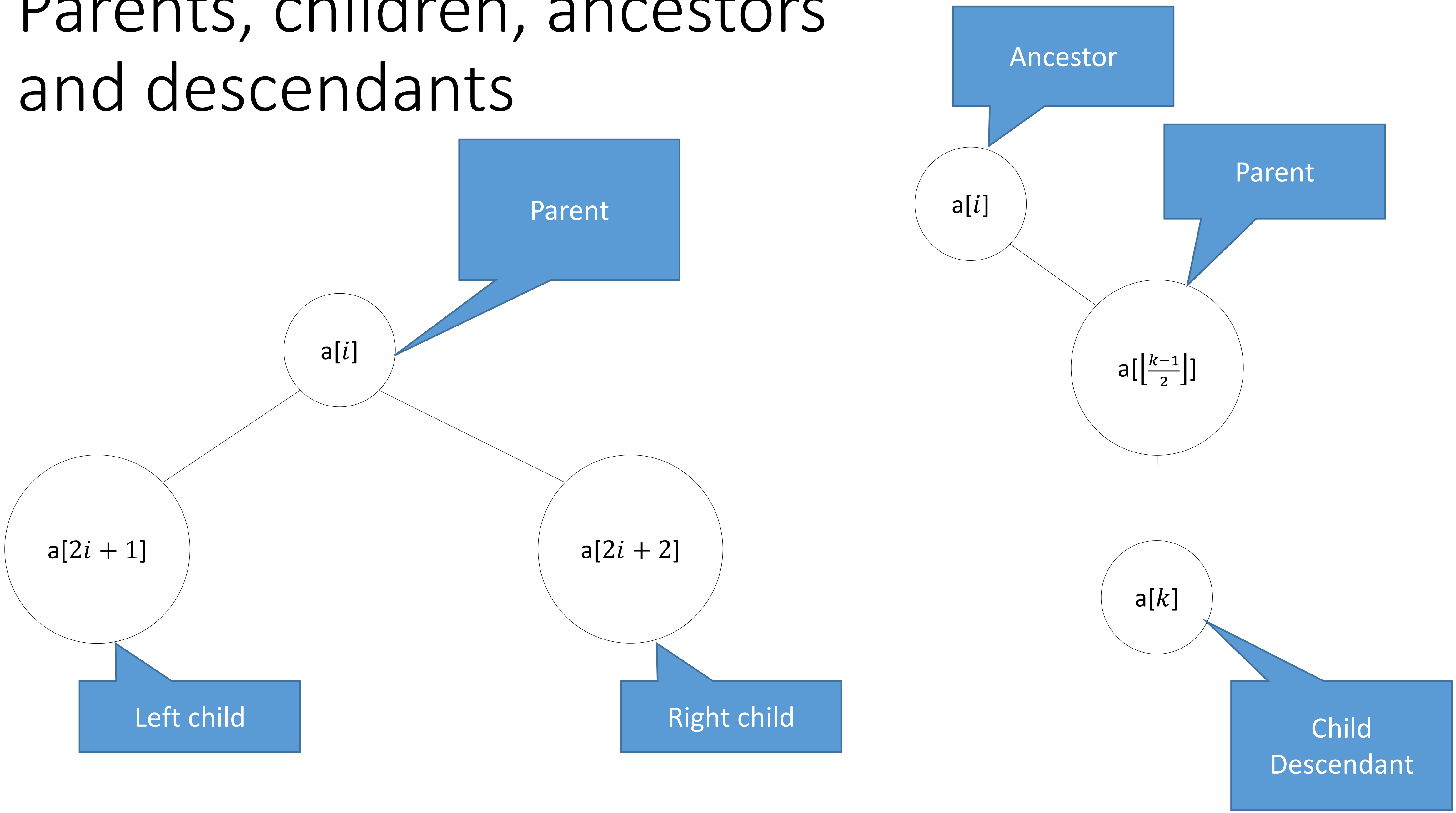
Heaps

See also Heap.dfy in Canvas

Arrays as binary trees



Parents, children, ancestors and descendants



Heap

- An array a **is a minheap** if and only if for each two indexes p and q within the array where a[p] is an ancestor of a[q] we have a[p]≤a[q]
- An array a **is a maxheap** if and only if for each two indexes p and q within the array where a[p] is an ancestor of a[q] we have a[p]≥a[q]
- If a is a minheap then the smallest value is in a[0]
- If a is a maxheap then the largest value is in a[0]
- A section within an array a fulfills the **minheap condition** if and only if for any two indexes p and q within the section where a[p] is an ancestor of a[q] we have a[p]≤a[q]
- A section within an array a fulfills the **maxheap condition** if and only if for any two indexes p and q within the section where a[p] is an ancestor of a[q] we have a[p]≥a[q]

Dafny predicate

```
predicate IsAncestorOf( p: int, q: int )
    decreases q;
    requires 0 <= p;
    requires 0 <= q;
{
     p < q && (p == (q-1)/2 || IsAncestorOf(p,(q-1)/2))
}
```

Dafny predicate

```
predicate IsMinHeap( a: seq<int>, i: int, j: int )
    requires 0 <= i <= j <= |a|;
{
    forall p,q | i <= p < q < j && IsAncestorOf(p,q) ::
        a[p] <= a[q]
}
```

Sorting and priority queues with heaps

- It is easy to use heaps to sort (heapsort)
- The most common method is to use a helper function that increases an area that fulfills a heap condition, a **rolldown** function
  - The function **rolldown** swaps a value down the tree until it in the correct position
- It is also possible to use two different helper functions, i.e. a **rolldown** function and also **rollup**, which increases the area in the other direction
  - The function **rollup** swaps a value up the tree (towards the root) until it is correctly positioned
- These two helper functions can also be used to implement priority queues with heaps

Dafny predicate

```
predicate IsMinHeapRollingDown
     ( a: seq<int>, i: int
     , k: int, j: int
     )
    requires 0 <= i <= k < j <= |a|;
{
    forall p,q |    i <= p < q < j &&
                    IsAncestorOf(p,q) &&
                    p != k ::
                        a[p] <= a[q]
}
```

This predicate is useful as a state description while rolling a value down the tree.
It is true if the area a[i..j] fulfills a heap condition except that the value in position k is perhaps too high up in the tree.

Dafny predicate

```
predicate IsMinHeapRollingUp
      ( a: seq<int>, i: int
      , k: int, j: int
      )
    requires 0 <= i <= k < j <= |a|;
{
    forall p,q |    i <= p < q < j &&
                    IsAncestorOf(p,q) &&
                    q != k ::
                        a[p] <= a[q]
}
```

This predicate is useful as a state description while rolling a value down the tree.
It is true if the area a[i..j] fulfills a heap condition except that the value in position k is perhaps too low in the tree.

Dafny lemma

```
lemma TransitiveAncestor( p: int, q: int, r: int )
    decreases r;
    requires 0 <= p;
    requires 0 <= q;
    requires 0 <= r;
    requires IsAncestorOf(p,q);
    requires IsAncestorOf(q,r);
    ensures  IsAncestorOf(p,r);
{
    if q == (r-1)/2 { return; }
}
```

Dafny lemma

```
lemma DescendantIsChildOrDescendantOfChild( p: int, q: int )
    requires 0 <= p;
    requires 0 <= q;
    requires IsAncestorOf(p,q);
    ensures q==2*p+1 ||
            q==2*p+2 ||
            IsAncestorOf(2*p+1,q) ||
            IsAncestorOf(2*p+2,q);
{}
```

Dafny lemma

```
lemma AllDescendantsAreChildrenOrDescendantsOfChild( p: int )
    requires 0 <= p;
    ensures forall q | q >=0 && IsAncestorOf(p,q) ::
        q==2*p+1 ||
        q==2*p+2 ||
        IsAncestorOf(2*p+1,q) ||
        IsAncestorOf(2*p+2,q);
{
    forall q | q>=0 && IsAncestorOf(p,q)
    {
        DescendantIsChildOrDescendantOfChild(p,q);
    }
}
```

Dafny lemma

```
lemma MinHeapRollingDownBecomesHeap( a: seq<int>, i: int, k: int, j: int )
    requires 0 <= i <= k < j <= |a|;
    requires IsMinHeapRollingDown(a,i,k,j);
    ensures 2*k+1 >= j ==> IsMinHeap(a,i,j);
    ensures 2*k+1 < j && 2*k+2 >= j && a[k] <= a[2*k+1] ==> IsMinHeap(a,i,j);
    ensures 2*k+2 < j && a[k] <= a[2*k+1] && a[k] <= a[2*k+2] ==> IsMinHeap(a,i,j);
{
    if 2*k+1 >= j { return; }
    if 2*k+1 < j && 2*k+2 >= j && a[k] <= a[2*k+1] { return; }
    if 2*k+2 < j && a[k] <= a[2*k+1] && a[k] <= a[2*k+2]
    {
        if IsMinHeap(a,i,j) { return; }
        // Given that IsMinHeap(a,i,j) is false, we will derive a contradiction
        var p,q :| i <= p < q < j && IsAncestorOf(p,q) && a[p] > a[q];
        DescendantIsChildOrDescendantOfChild(p,q);
        assert false; // Yes! We have a contradiction, so IsMinHeap(a,i,j) is true
    }
}
```

Dafny lemma

```
lemma ZeroIsRoot( p: int )
    decreases p;
    requires p > 0;
    ensures IsAncestorOf(0,p);
{
    if p == 1 || p == 2 { return; }
    ZeroIsRoot((p-1)/2);
    TransitiveAncestor(0,(p-1)/2,p);
}
```

Dafny lemma

```
lemma RootHasMin( a: seq<int>, i: int, j: int )
    requires 0 <= i <= j <= |a|;
    requires IsMinHeap(a,i,j);
    ensures forall p | i <= p < j && IsAncestorOf(i,p) :: a[i] <= a[p];
{}
```

Dafny lemma

```
lemma ZeroHasMin( a: seq<int>, n: int )
    requires 0 < n <= |a|;
    requires IsMinHeap(a,0,n);
    ensures forall p | 0 < p < n :: IsAncestorOf(0,p);
    ensures forall p | 0 <= p < n :: a[0] <= a[p];
{
    forall p | 1 <= p < n { ZeroIsRoot(p); }
}
```

Dafny lemma

```
lemma PartOfHeap( a: seq<int>, i: int, j: int, p: int, q: int )
    requires 0 <= i <= p <= q <= j <= |a|;
    requires IsMinHeap(a,i,j);
    ensures IsMinHeap(a,p,q);
{}
```